(AT) SCOII2WD - CHARLES et al.

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : H01L 21/00 — A2

(11) International Publication Number: **WO 99/31713**

(43) International Publication Date: 24 June 1999 (24.06.99)

(21) International Application Number: PCT/IB98/02147

(22) International Filing Date: 10 December 1998 (10.12.98)

(30) Priority Data:

60/069,034	12 December 1997 (12.12.97)	US
09/207,957	9 December 1998 (09.12.98)	US

(81) Designated States: JP, KR, SG, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
Without international search report and to be republished upon receipt of that report.

(71) Applicant: JENOPTIK AG [DE/DE]; Carl-Zeiss-Strasse 1, D-07739 Jena (DE).

(72) Inventors: ELLIS, Raymond, W.; 11220 Pinehurst Drive, Austin, TX 78747 (US). EXTINE, Andrew, S.; 16507 Old Stable Road, San Antonio, TX 78247 (US). HARDEE, Bedford, Eugene; 2212 Valley View Drive, Woodland Park, CO 80863 (US). LEMCHAK, Michael, Richard; 2095 Tyrone Drive, Colorado Springs, CO 80919 (US). SIMMONS, Michael, C.; 6305 Gibson Drive, Orlando, FL 32809 (US).

(74) Agent: GEYER, Werner; Perhamerstrasse 31, D-80687 München (DE).

(54) Title: INTEGRATED MATERIAL MANAGEMENT MODULE

(57) Abstract

A module for use in a system for processing articles, in which the system includes a plurality of machine tools for processing articles, a pod for carrying the articles to be processed by the machine tools from one machine tool to another, a host processing controller associated with the machine tools for controlling the operation thereof, a robot connected to each machine tool for receiving a pod, opening the pod and for transporting the articles from within the pod into position on the machine tool for processing. An identification means is carried by the pod for identifying a particular pod and the articles carried in the pod. The module includes a single wire connection between the identification means, the host controller and the robot. The module has microprocessor means to identify the source of a signal and means for routing the signal between the identification means and the host controller and between the host controller and the robot depending on its source.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

INTEGRATED MATERIAL MANAGEMENT MODULE

FIELD OF THE INVENTION

The present invention relates generally to the field of manufacturing systems, and more particularly to a stand alone unit or module used to facilitate the interface or communication between elements in a system for processing articles, such as semiconductor wafers.

BACKGROUND OF THE INVENTION

The field of semiconductor wafer manufacturing has seen the development of technologies which minimize wafer contamination, thus maximizing yields, through the use of sealed pods or containers for transporting wafers to be processed from one processing tool to another and to and from storage areas. Access to the wafers within the containers is had through a standard mechanical interface (SMIF). For example, see U.S. Patent 4,532,970. Fabrication systems using such pods have come to be known as SMIF fabs and the containers have come to be known as SMIF pods.

The system for processing wafers in a SMIF fab will typically include a SMIF pod for carrying the wafers, a machine or processing tool (having a processing controller - referred to as the "host") for processing the wafers, a robot which is connected to the machine tool for receiving the pod and for transporting the wafers from within the pod into position on the machine tool for processing, and an identification system to permit identification of a pod. The identification system will consist of an identifying tag on the pod, such as a bar code or RF transponder, and a reader. The type of reader will depend on the type of identifying tag.

In a typical existing installation the identification system would be connected to a Cell Controller which manages the process tools via a terminal server or similar hardware and then via separate connections to the robot at each tool port. In addition, the Cell Controller would be separately connected with the process tools and their associated robots.

This arrangement is complicated since there are numerous tools within a fab and for each tool and its associated robot there are various data collection points for monitoring the environment and the articles being processed. Therefore a number of different data inputs is required. Currently, a separate computer is required to coordinate the collection of these inputs. In addition, for each of the numerous tools in a fab there is the problem of determining where to locate antennae, controllers and power supplies required to effect proper communication. Positioning of the antennae and other components significantly complicates the design of any fab.

Further, the use of an additional computer to effect communication requires means of interfacing with such computer and integrating with the various tools.

OBJECTS OF THE INVENTION

It is accordingly a principal object of the present invention to overcome the foregoing difficulties and disadvantages.

It is a specific object of the invention to simplify the current connection arrangement, discussed above, by providing a single wire interface between a process tool, the identification system, and the robot of that tool to the Cell Controller. Accordingly, this invention provides a module which includes a reader for reading the identification tag, and a micro circuit to provide a connection route between the identification system and the host, and between the host and the robot.

Another object of the invention is to provide a simplified connection using lower power consumption and modular design so that it can be easily installed.

Other objects, features and advantages of the present invention will be apparent from the description hereinafter.

SUMMARY AND BRIEF DESCRIPTION

In a semiconductor wafer processing system, the identification system could be of the typical RF reader and transponder type, or of a bar code type, or of an IR transceiver type. The robot could be of a well know structure for receiving the pod of wafers and moving the wafers into position where the process tool will have access to the pod for treating the wafers. The host can either be integral with the tool or may consist of a computer network for providing control and instruction to each of the tools.

In operation, the RF Reader of the identification system (or in the case of a bar code the optical reader, or in case of an IR system, the IR transceiver) would be mounted on or in the robot in such a manner as to permit reading the identification tag when the pod is properly placed on a receiving port of the robot. The tag will be read upon command of the host upon receipt of a message from the robot that a pod has been placed for processing. All messages and commands from the host via the present invention are simply passed to the proper recipient without respect to the contents of the message or command. The module of this invention does not provide message processing, only message routing. Once the host has determined that the pod is in position it will then communicate through the module to the robot to instruct the robot to begin its operation on the pod allowing the process tool access to the wafers. The module does not communicate directly with the

pod or robot, either through the identification system or other means, and does not require any type of data storage or processing capability on the pod to facilitate additional process requirements of the wafers contained within the pod.

The present invention thus operates as a distribution system. It is capable of recognizing the source of information and routing it to the proper recipient. For example, the signal coming from the identification system would be routed to the host. Similar signals that may come from the robot to indicate that it has received the pod would also be routed to the host, and signals directed to the robot from the host to perform certain functions would be routed through the module.

The present invention uses lower power to effect communication, is faster and less expensive than current techniques, and is a stand alone module that can be easily installed in a variety of fabs. The use of this invention does not require reconfiguration of robots or tools and it can interface with many different existing robots produced by different manufacturers.

In addition to the foregoing, the present invention can interface with guided vehicles used in a fab for transporting wafer containers.

The foregoing and other features of the present invention are more fully described with reference to the following drawings annexed hereto.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a hardware diagram of the functional blocks of a circuit card assembly for a microcomputer based module according to the present invention;

Fig. 2 shows the connection of the integrated material management module of the present invention with existing hardware components;

Fig. 3 shows the communications link for a guided vehicle;

Fig. 4 is the overall software calling graph of the integrated material management module code;

Fig. 5 shows the message traffic for a robot to host communication scenario;

Fig. 6 is the state diagram associated with a message from the robot to the host for SECS interface;

Fig. 7 shows how an unknown state is resolved;

Fig. 8 shows the message traffic for a host to robot scenario;

Fig. 9 shows the message traffic for a host to the module scenario;

Fig. 10 is the state diagram associated with a message from the host;

Fig. 11 shows the message traffic for the module to host scenario;

Fig. 12 is the state diagram associated with a message from the module; and

Fig. 13 is the state diagram associated with an ASCII interface.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

With reference to the drawings, the following is a non-limiting description of various preferred embodiments of the invention.

Fig. 1 shows the functional blocks of a circuit card assembly 20 (or CCA) for a microcomputer based Integrated Material Management Module (IMM) 10. Fig. 2 shows that the IMM module 10 provides the connection with an existing host-robot-vehicle-identification message stream.

Because the technology herein is equally suitable for processing products of types different than semiconductor wafers, the semiconductor industry is hereinafter discussed for illustrative and descriptive purposes, and the term "wafer" is used hereinafter to designate material, or an article, of any equally suitable product. With this is mind, several different embodiments of the invention will be described.

Referring to Fig. 1, in different preferred embodiments of the invention, the host 11 could be a Manufacturing Execution System (MES) such as the "Factory Works" product produced by FASTECH, Inc. or the host could be an Equipment Set Controller (ESC) possibly based on the Tool Object Model (TOM) or the host could be the tool, itself.

In different preferred embodiments of the invention, the robot 12 could also be an elevator or a loader. Still, referring to Fig. 1, in different embodiments of the invention, the identification system 13 could also be called a reader or a tracker and could be based on a system such as the Iridnet Advanced Tracking System sold by Jenoptik Infab, Inc. The delivery vehicle 14 could be a person guided Vehicle (PGV) or a Rail Guided Vehicle (RGV) or an Automated Guided Vehicle (AGV). In yet another embodiment, the delivery vehicle 14 may be absent, replaced with an operator's manual delivery of wafer carriers.

The IMM circuit card assembly 20 includes the following functional blocks:

Power Distribution, management, and control

The CCA 20 is powered by a single 24vdc source 15. This is then regulated to supply 5vdc 18 for the logic components (microprocessor 16, memory, etc.); a separate 5vdc supply 17 which is

switched at a frequency greater than 200 khz (to avoid harmonic interference) for the barcode and radio frequency identification readers, and 15vdc supply 19 for the Infrared Transceiver identificaiton reader 21. In addition, the circuit is provided with a reset control device to insure that upon the applicaiton of power or a device reset that all of the components are properly initialized to insure a deterministic state for program initialization.

Microprocessor and memory

The high speed microprocessor 16 is provided with a read only memory (ROM) for the purpose of initialization, program maintenance, and program loading. In addition to the read only memory, random access memory for program storage and operation is provided in a non-volatile form, i.e. the memory contents are preserved in the absence of power. The design also provides a "mode selection" control where upon initialization the program can determine the communications protocol of the host system.

Communications

The design provides electrical communications interfaces to the host system 11 and to robotic loading devices 12 via industry standard RS232 serial interface 22. All interfaces to the host equipment, loading services, and reader are under program control.

Communications to the barcode and/or RF reader 13 is via an industry standard RS232 interface 23. This interface is configurable for data rate and other serial interface 22. All interfaces to the host equipment, loading devices, and reader are under program control.

Communications to the barcode and/or reader 13 is via an industry standard RS232 interface 23. This interface is configurable for data rate and other serial interface parameters as required for the receiving device. Power for these devices is provided from the power management and distribution section. By providing power directly to the devices the proper power requirements can be maintained. In addition to the barcode and RF readers, communications support for an Infrared Transceiver 21 is provided. This device uses a non-standard serial communications methodology where a single wire 24 is used for both transmit and receive whereas in a standard RS-232 serial interface two wires are used - one for transmit and one for receive. In conjunction with this non-standard serial communications methodology the power requirements of the Infrared Transceiver are met by the power management and distribution design 19.

Power up/reset:

Upon an initial power-up condition, a reset via switch S1, or a reset under microprocessor control, the microprocessor Reset Control circuit, maintains a reset state for approximately 350 milliseconds (ms) to allow power, the microprocessor, and external components to stabilize prior to initialization.

Bootstrap/Program Load:

Upon expiration of the reset timer the microprocessor begins program and hardware initialization tasks. The "bootstrap" program first initializes the microprocessor 16 to local control from internal read only memory (ROM) to perform the hardware initialization of the serial port that is required for program load. Once the "loader"/loader serial port is initialized the bootstrap program

checks the serial port t determine if a Device Transmit Ready (DTR) signal is present. The presence of a DTR signal indicates that the user/host intends to download a new or updated Operational Program. The absence of a DTR signal indicates that the existing Operational Program is to be executed. If the bootstrap program determines that the existing Operational Program is to be executed it executes a program code segment to enable the external memory device containing the Operational Program and program execution resumes with the Operational Program.

In the event that the bootstrap program detects that a new or updated Operational Program is to be loaded, control is passed to the program load routine which will then interface with the users terminal device to control the downloading and verification of the new/updated Operational Program. Upon completion of this process and the removal of the DTR signal and/or the users terminal device the microprocessor, under initial program control, will initiate a device reset to allow control to pass to newly installed Operational Program.

Mode selection:

The invention includes a discrete switch that allows selection of different modes of program operation by the Operational Program. Upon program initialization the Operational Program reads the microprocessor port that the mode control switch is attached to. After performing this function the Operational Program initializes to the proper mode and normal operation begins:

The following modes are supported by the design:

Mode-0	SECs Messaging with RF ID reader
Mode-1	SECs Messaging with Barcode ID reader
Mode-2	SECs Messaging with Infrared ID reader

Mode-4	ASCII Messaging with RF ID reader
M de-5	ASCII Messaging with Barcode ID reader
Mode-6	ASCII Messaging with Infrared ID reader

Modes 3 and 7 are reserved for future use.

Below is a (1) System Overview; (2) a Hardware Overview; (3) a Software Overview; (4) a description of the Communication Control for a SECS Interface; and (5) Communication Control for an ASCII Interface.

In the following description, reference is made to code designations, identified and described in the "APPENDIX." The APPENDIX also includes descriptions of S3F1 message processing; the RF ID Interface; the ID Port Handlers; the Serial Port Handlers; the Millisecond Timer Handlers; the Watchdog Timer Handlers; and the Executive Routine.

1. System Overview

With reference to Fig. 2, the integrated MaterialID System consists of an ID system 13, such as an RF ID system, and a specialized micro-controller interface that provides the connection with the existing host-robot (11-12) message stream. The micro-controller, or Integrated MaterialID Module (IMM), may be configured to accommodate either SECS or ASCII messages. The IMM 10 is positioned between the host 11 and the robot or elevator 12.

The IMM module 10 passes messages from the host 11 to the robot 12 and back, unless the message is a material id message from the host. When it receives a material id message, it intercepts the message, reads the material id from the ID system 13, and generates an appropriate response. Thus, the robot 11, IMM 10, and the ID system 13 together present an integrated material identification capability to the host through the single wire connected to IMM 10.

2. Hardware Overview

The Integrated MaterialID System hardware consists of a micro-controller-based IMM board, a separate RF ID system, and a power supply. The IMM is mounted in the robot controller card cage, and connections are made through the front panel. The RF ID system consists of an integrated RF transmitter and antenna, and is mounted on the base plate of the robot, next to the SMIF pod. It is connected to the micro-controller by a flexible cable that has an RS-232 interface and 5V power. The power supply is separate from both the micro-controller and the RF ID system, and is mounted in the robot chassis outside the card cage. Packaging the RF transmitter and antenna in a separate module from the micro-controller avoids introducing RF into the existing robot control system, and it provides the capability of reusing the RF ID system in other applications.

The IMM module uses an 8051 compatible Dallas 87C520 microprocessor. The microprocessor has two built-in serial ports 25 and 26, 16 Kbytes of one-time-programmable internal read only memory (EPROM), and 128 Kbytes of external data memory configured as non-volatile memory through the use of Lithium battery backup which affords up to 10 years of program memory retention. In addition to the built-in serial ports, two additional serial ports are available to supply communications to an Identification System using industry standard RS232 communications (such as bar-code or RF readers) 13, a propriety interface to support the Infra-red Transceiver (IRT) component 21 of Infab's IridNet Tracking System, and two additional ports (14 for the vehicle and 14a) to support IRDA infra-red serial communications for providing command and control communications for semiconductor wafer transport devices such as push carts, rail-guided vehicles, and automated guided vehicles in conjunction with and coordinating with future automated overhead

transp rts systems used for placing semiconductor wafer carriers directly on the semiconductor workstations loadport.

The RF ID system uses a Texas Instruments TIRIS micro-reader (RI-STU-MRD1) and a 45 micro-Henry wire loop antenna. This reader was chosen for its small size, low power consumption, low radiated energy, and low cost. The entire unit is assembled as one external package, and it is connected to the micro-controller through an RS-232 interface. The connecting cable supplies regulated power (5V) as well.

Both the IMM and the RFID system are powered by a 5V linear power supply, separate from the robot power supplies. This was done to avoid possible interaction between the switching power supplies in the robot and the RF circuity, though in production it may prove desirable to use power directly from the robot control system.

3. Software Overview

The general structure of the IMM code is that of a communication controller state machine driven by events from the serial ports and the millisecond timer clock. The ID system is built as a layer on one of the serial ports. The finite state communication control may be configured to handle either SECS or ASCII messages (though not both at once!). Currently, this configuration is done through conditional compilation which controls the initialization code, though in production except for initialization takes place at interrupt level; the background task in the executive does nothing. A hardware watchdog timer runs in the background and will reset the system if it appears to be hung. The overall calling graph (who calls whom) is depicted in Fig. 4.

In the SECS configuration, communication with the h st and the rob t is controlled by the SECS communication control module 27, a finite state machine that recognizees the ENQ-EOT-MSG-ACK sequence of SECS messages. Its primary purpose is to recognize and block S3F1 (Material Status Request) messages sent by the host, and to keep track of the state of the host-robot communication so that the IMM does not attempt to send a reply in the middle of a host-robot conversation. Incoming SECS messages are buffered, and if the message is an S3F1, it is processed in the S3F1 module 29, which invokes routines from the ID system 13. The ID system starts an RF system read, detects completion, and invokes a callback routine in the S3F1 module 29 to generate the S3F2 (Material Status Data) response.

In the ASCII configuration, all message processing takes place in the ASCII communication control module 28. This is a small finite state machine that follows the ASCII communication protocol documented in the various Infab service manuals. A message arriving from the host is read, buffered, and inspected to see whether it is a READID material id message. If it is not, the message is sent to the robot. The module then waits for a reply from the robot, buffers it, and sends it to the host. If the message from the host is a material id message, a request is made of the ID system to start a read. When it completes, the ID system invokes a callback to send a READID reply to the host.

The files that compose the system are:

- reg80c320.h
- exec.h / exec.c
- SECS.h / SECS.c
- S3F1.h / S3F1.c
- ASCII.h / ASCII.c
- IDSys.h / IDS.ys.c
- IDPort.h / IDPort.c

- msTimer.h / msTimer.c
- wd.h / wd.c
- serialPorts.h / serialPorts.c

and they are generated from their respective .html files.

4. Communication Control for SECS Interface

The communication control module is command center for IMM activity. Its general purpose is to pass messages back and forth from the host to the robot and back, intercepting and acting on material ID requests. It must recognize a material ID request, so that it can initiate a read from the ID system; and it must recognize when the host and robot are engaged in communication, so that it does not attempt to send the reply to a material ID request in the middle of their conversation.

Design Considerations

Two general approaches to solving this problem were considered. One was to buffer the SECS-I message in the IMM and forward it to its final destination only after the entire message had arrived and checked. At first this appeared to be a simple, straightforward approach, but it introduces complicated problems trying to simulate the behavior of the normal host-robot system in failure situations. For instance, what if the IMM receives a message from the host, and then discovers that the robot is not responding? How can it notify the host? Without the IMM, the host would know immediately that the robot was down: the robot would not even respond to the ENQ-EOT line negotiation protocol. The IMM responded before it even discovered the robot was down, though, so the host will be confused.

The approach adopted in the IMM, however, is to forward the message a byte at a time, as soon as it arrives. This requires almost no buffering at all, and the host and robot can maintain an accurate picture of status of the link, as if there were no IMM at all. Keeping track of the status of the communication link, however, is somewhat more complex. We divide message traffic into robot-initiated, host-initiated, and IMM-initiated messages, and discuss them below.

Message from Robot

Consider the case of a message from the robot for the host. Perhaps it is the response to a host inquiry, or an event report directed to the host. The message traffic for this scenario is depicted in Fig. 5

MSG is just a shorthand for the stream of bytes that make up the message — each byte of the message is still forwarded as soon as it arrives. The boxes along the IMM line show the state of the communication controller, which allows it to recognize when the host-robot link is idle. The complete state diagram associated with a message from the robot to the host is shown in Fig. 6.

Note the contention resolution (state ENQ_robot transitioning to ENQ_host upon receiving an ENQ from the robot), which occurs if both the host and the robot attempt to initiate a conversation at once. Note, also, that all timeouts return the controller to an IDLE state.

UNKNOWN state

Fig. 7 introduces the UNKNOWN state, the state the IMM starts in, and the state it returns to when it encounters an unexpected sequence of events. When the IMM is in the UNKNOWN state, it simply forwards every byte it receives, and tries to re-sync. It does not initiate any message of its

wn, lest it interrupt a host-robot conversation in progress. Although in normal message traffic this does not occur, if the host or the robot is restarted or a substantial interruption in the link to one of them occurs, an unintelligible sequence can result. This transitions the IMM into an UNKNOWN state, which is resolved as depicted in Fig. 7.

One basic starting point the IMM uses to recognize the status of the link is passage of time: if neither the host nor the robot have transmitted anything for a T2 period, it is safe to assume there is no conversation in progress and that the link is IDLE. The other starting point is the transmission of an ENQ. This is the start of a normal negotiation sequence, and the expected response is an EOT. If this does not occur (say one end was just re-started, and the other has not stumbled to the fact yet), the IMM re-enters the UNKNOWN state. If both the host the robot follow the SECS-I protocol, the IMM will eventually transition out of the UNKNOWN state. The use of an UNKNOWN state allows the IMM to accommodate the restart of the host or the robot (or the IMM) in a graceful way, to eventually recognize the state of the communication link between them, and not to interrupt the normal flow of message traffic while it is doing this.

Message from Host

Now the consider the case of a message from the host to the robot. The message traffic for this scenario is shown in Fig. 8.

The decision to forward the message a byte at a time, however, introduces a problem: when the host sends a material ID request, how can the IMM know not to send it on to the robot, until it has already sent most of it? The solution used in the IMM is to send the message on to the robot, as normal, but to change the final byte of the checksum to an incorrect value, see Fig. 9.

Note again the contention resolution (state ENQ_robot transitions to ENQ_host upon receiving an ENQ from the robot), and the fact that all timeouts return to an IDLE state.

Message from IMM

Finally, consider how the IMM sends a message ID reply to the host. This is a simple SECS-I sequence, shown in Fig. 11.

The robot will behave as if the message has been corrupted, send back a NAK, and take no further action. The IMM, however, is aware that the message has been received correctly, and returns an ACK to the host. Note that if the message is corrupted in any way, say with a bad checksum, it will not be recognized by the IMM as a material ID request and it will be passed on through to the robot, where it will be treated as any other unrecognized message. This whole approach relies on the robot's ability to reject corrupted messages, of course, but since that is a basic part of the SECS-I standard, the approach works reliably.

A state diagram that is associated with both these sequences is shown in Fig. 10. Characters arriving from the robot are ignored until the message has been sent and acknowledged. Since under normal circumstances this only takes a couple of dozen milliseconds (the message is ten characters long, and one character takes about a millisecond at 9600 baud), the robot only experiences a slight delay, considerably less than the T2 period that it will tolerate. If the host does not respond to the

IMM message and a T2 timeout occurs, then the robot will experience a timeout, too, but that would probably have occurred if the IMM had been busy sending a message anyhow. The state diagram assocaited with this sequence is shown in Fig. 12.

Note that the IMM will try to re-send the material ID reply if the host fails to acknowledge it the first time. The number of retry attempts, however, currently defaults to 0, in agreement with the current robot SECS interface definition. In this configuration, all timeouts return the IMM to an IDLE state.

5. Communication Control for ASCII Interface

The module is responsible for the coordination of all communication with the external devices -- the host, the robot, the ID system -- or the ASCII message protocol configuration. It registers itself during initialization as the handler for all communication with the robot and host serial ports, and it communicates with the ID system through the ID read interface. Like the SECS communication control, it implements a finite state machine, but because it reads and buffers an entire message at every step and does not deal with it a character at a timer, the state machine is less complicated, see Fig. 13.

After initialization, the module waits to receive a message in the RECEIVING_FROM-HOST state. The receiveHost routine saves the incoming character sequence in a buffer, and upon receiving a or it inspects the message and proceeds to the next step. If the message is a normal robot command, not a material id message, enters the SENDING_TO_ROBOT state and starts transmitting the message to the robot serial port. When sendRobot has sent the message, it waits for the robot to send a reply in the RECEIVING_FROM_ROBOT state. When an entire message has been

received from the robot, receiveRobot enters the SENDING_TO_HOST state and starts transmitting the message on the host serial port. When the message has been sent, sendHost returns to the RECEIVING_FROM_HOST state and begins the cycle again.

If the message is a material id request, "READID," the finite state machine enters the READING_ID state and initiates a read from the ID system. When the ID system completes the read, it signals the ASCII module by invoking the callback routine with the status and data. This routine formats a reply message and sends it to the host.

Notice that a character can arrive from either source, ROBOT or HOST, at any time, and the ASCII module must be prepared to deal with unexpected input. This is particularly simple, for there is exactly one event associated with each state, and one handler for each event. Thus the routines just checks to see that the module is in the state for which they are expected to be active, and if they are invoked under other circumstances, they simply ignore the input and return.

There is a safety net built in to the ASCII module to help it avoid getting locked up. If, for instance, the terminating in a command should get lost or garbled, the module could wait forever for a message to complete. This is done by running the watchdog timer in the background, and clearing it whenever the module completes a cycle and re-enters the RECEIVING_FROM_HOST state. Since we don't want the watchdog to go off just because a message hasn't arrived for a while, though, we also use the millisecond timer facility to periodically run a timeout routine, which clears the watchdog if we are just idle with an empty buffer.

APPENDIX

Communication Control for SECS Interface

The communication control module is command center for IMM activity. Its general purpose is to pass messages back and forth from the host to the robot and back, intercepting and acting on material ID requests. It must recognize a material ID request, so that it can initiate a read from the ID system; and it must recognize when the host and robot are engaged in communication, so that it does not attempt to send the reply to a material ID request in the middle of their conversation.

Implementation

Given the description of the state machine above, the code is quite simple, if tedious. The complete list of states is

UNKNOWN
: link status not determined; see discussion

IDLE
: determined that no conversation is in progress

ENQ_robot
: received ENQ from host, forwarded to robot, awaiting EOT from robot

EOT_host
: received EOT from robot, forwarded to host, awaiting LEN byte from host

MSG_robot
: received a message character from host, forwarded to robot, awaiting next character in message

CHECK_robot
: received last checksum character from host, forwarded to robot, awaiting ACK/NAK

BADCHECK_robot
: received last checksum character of ID request message from host, sent corrupted checksum to robot, awaiting NAK

ENQ_host
: received ENQ from robot, forwarded to host, awaiting EOT from host

EOT_robot
: received EOT from host, forwarded to robot, awaiting LEN byte from robot

MSG_host
: received a message character from robot, forwarded to host, awaiting next character in message

CHECK_host
: received last checksum character from robot, forwarded to host, awaiting ACK/NAK

IDENQ_host
: sending message ID reply, sent ENQ to host, awaiting EOT

Communcation controller for SECS interface

IDMSG_host
received EOT, sent message character to host; if message finished, awaiting ACK/NAK

In general, every state is associated with three procedures:

- XXX_enter, a procedure that is executed upon entering the state
- XXX_proc, a procedure to process events while in that state
- XXX_timeout, a procedure to handle timeouts while in that state

For example, there is EOT_robot_enter, EOT_robot_proc, and EOT_robot_timeout. There is a table of the event processing procedures, indexed by state, which is used by the interrupt service routines to deliver events.

There is only one externally visible service, SECS_sendMsgToHost, to send a message to the host when the communication channel is free. It is used by the S3F1 processing module to reply to a message ID request. It stores away a pointer to the message to be sent and, if the link is currently IDLE, invokes IDENQ_host_enter to start sending the message. If the communication link is busy, the next time IDLE_enter is called, it will check to see if there is a message to be sent and, if so, invoke IDENQ_host_enter to start sending it.

Watchdog Timer

As in the ASCII interface communication controller, there is a safety net built in to the SECS communication controller to help it avoid getting locked up. This is done by running the watchdog timer in the background, and clearing it whenever the module completes a cycle and re-enters the IDLE state. Since we don't want the watchdog to go off just because a message hasn't arrived for a while, we also use the millisecond timer facility to periodically run a timeout routine, which clears the watchdog if we are currently IDLE.

SECS.h

```
/*
  Communcation controller for SECS interface
  */

#ifndef _SECS_H_
#define _SECS_H_

extern void SECS_init(void);

extern unsigned char SECS_sendMsgToHost(
    unsigned char length,
    unsigned char *ptr);

#endif /* _SECS_H_ */
```

SECS.c

```
#include "exec.h"
#include "wd.h"
```

Communcation controller for SECS interfac

```
#include "serialPorts.h"
#include "msTimer.h"
#include "S3F1.h"

#include "SECS.h"

/*
  Default SECS parameter values, compatible with Ergospeed 3x00
  */
#define DEFAULT_T1_ms       500
#define DEFAULT_T2_ms     10000
#define DEFAULT_MaxRetry      0

unsigned short T1_ms     = DEFAULT_T1_ms;
unsigned short T2_ms     = DEFAULT_T2_ms;
unsigned char  MaxRetry  = DEFAULT_MaxRetry;

/* Control character values */
#define EOT   0x04
#define ENQ   0x05
#define ACK   0x06
#define NAK   0x15

/* forward declarations */
static void robotReceive(uchar val);
static void hostReceive(uchar val);
static void watchdog_timeout(uchar dum);
static void passThrough(uchar src, uchar val);

static void UNKNOWN_enter();
static void UNKNOWN_proc(uchar src, uchar val);
static void UNKNOWN_timeout(uchar id);

static void IDLE_enter();
static void IDLE_proc(uchar src, uchar val);

static void ENQ_robot_enter();
static void ENQ_robot_proc(uchar src, uchar val);
static void ENQ_robot_timeout(uchar id);

static void EOT_host_enter();
static void EOT_host_proc(uchar src, uchar val);
static void EOT_host_timeout(uchar id);

static void MSG_robot_enter(uchar len);
static void MSG_robot_proc(uchar src, uchar val);
static void MSG_robot_timeout(uchar id);

static void CHECK_robot_enter();
static void CHECK_robot_proc(uchar src, uchar val);
static void CHECK_robot_timeout(uchar id);

static void BADCHECK_robot_enter();
static void BADCHECK_robot_proc(uchar src, uchar val);
static void BADCHECK_robot_timeout(uchar id);

static void ENQ_host_enter();
static void ENQ_host_proc(uchar src, uchar val);
static void ENQ_host_timeout(uchar id);

static void EOT_robot_enter();
static void EOT_robot_proc(uchar src, uchar val);
static void EOT_robot_timeout(uchar id);
```

Communcation controller for SECS interface

```
static void MSG_host_enter(uchar len);
static void MSG_host_proc(uchar src, uchar val);
static void MSG_host_timeout(uchar id);

static void CHECK_host_enter();
static void CHECK_host_proc(uchar src, uchar val);
static void CHECK_host_timeout(uchar id);

static void IDENQ_host_enter();
static void IDENQ_host_proc(uchar src, uchar val);
static void IDENQ_host_timeout(uchar id);

static void IDMSG_host_enter();
static void IDMSG_host_proc(uchar src, uchar val);
static void IDMSG_host_timeout(uchar id);

static void SECSMSG_init(void);
static void SECSMSG_proc(uchar inChar, uchar remain, uchar *abortFlag);
static void SECSMSG_ack(void);

/* Globals */

/* States */
enum
{
    UNKNOWN = 0,
    IDLE,
    ENQ_robot,        /* host-robot sequence */
    EOT_host,
    MSG_robot,
    CHECK_robot,
    BADCHECK_robot,   /* end host-IMM sequence */
    ENQ_host,         /* robot-host sequence */
    EOT_robot,
    MSG_host,
    CHECK_host,
    IDENQ_host,       /* IMM-host sequence */
    IDMSG_host,
    MAX_STATE         /* sentinel */
};

/* Branch table -- handler for each state */

typedef void (*handlerProc)(uchar, uchar);

code handlerProc handler[] =
{
    UNKNOWN_proc,
    IDLE_proc,
    ENQ_robot_proc,
    EOT_host_proc,
    MSG_robot_proc,
    CHECK_robot_proc,
    BADCHECK_robot_proc,
    ENQ_host_proc,
    EOT_robot_proc,
    MSG_host_proc,
    CHECK_host_proc,
    IDENQ_host_proc,
    IDMSG_host_proc
};

static uchar state = UNKNOWN;
```

Communcation controller for SECS interface

```
#define BUF_SIZE 256
static xdata uchar buf[BUF_SIZE];
static uchar bufIndex;
static uchar charRemain;
static uchar timerID;
static uchar msgLen;
static uchar *msgPtr;
static uchar tryCount;

void
SECS_init(void)
{
    msgPtr = NULL;

    SER_setHandler(ROBOT, RECEIVE, robotReceive);
    SER_setHandler(HOST, RECEIVE, hostReceive);

    UNKNOWN_enter();

    watchdog_timeout(0);
}

static void
robotReceive(uchar val)
{
    (handler[state])(ROBOT, val);
}

static void
hostReceive(uchar val)
{
    (handler[state])(HOST, val);
}

bool
SECS_sendMsgToHost(uchar length, uchar *ptr)
{
    if (msgPtr != NULL)
        return FALSE;

    msgLen = length;
    msgPtr = ptr;
    tryCount = 0;

    if (state == IDLE)
        IDENQ_host_enter();

    return TRUE;
}

static void
watchdog_timeout(uchar dum)
{
    if (state == IDLE)
        WD_reset();

    MST_create(10000, watchdog_timeout);
}

static void
passThrough(uchar src, uchar val)
{
    if (src == ROBOT)
        SER_put(HOST, val);
```

Communcation controller for SECS interfac

```
    else if (src == HOST)
        SER_put(ROBOT, val);
}

static void
UNKNOWN_enter()
{
    state = UNKNOWN;
    timerID = MST_create(T2_ms, UNKNOWN_timeout);
}

static void
UNKNOWN_proc(uchar src, uchar val)
{
    passThrough(src, val);
    MST_cancel(timerID);

    if (val == ENQ)
    {
        if (src == HOST)
            ENQ_robot_enter();
        else if (src == ROBOT)
            ENQ_host_enter();
    }
    else
        UNKNOWN_enter();
}

static void
UNKNOWN_timeout(uchar id)
{
    if (timerID == id)
        IDLE_enter();
}

static void
IDLE_enter()
{
    WD_reset();

    if (msgPtr != NULL)
        IDENQ_host_enter();
    else
        state = IDLE;
}

static void
IDLE_proc(uchar src, uchar val)
{
    passThrough(src, val);

    if (val == ENQ)
    {
        if (src == HOST)
            ENQ_robot_enter();
        else if (src == ROBOT)
            ENQ_host_enter();
    }
    else
        UNKNOWN_enter();
}

static void
ENQ_robot_enter()
```

Communcation controller for SECS interface

```
{
    state = ENQ_robot;
    timerID = MST_create(T2_ms, ENQ_robot_timeout);
}

static void
ENQ_robot_proc(uchar src, uchar val)
{
    passThrough(src, val);

    if (src == ROBOT)
    {
        if (val == EOT)
        {
            MST_cancel(timerID);
            EOT_host_enter();
        }
        else if (val == ENQ)
        {
            MST_cancel(timerID);
            ENQ_host_enter();
        }
        else
            /* wait for another character from robot */ ;
    }
    else
    {
        MST_cancel(timerID);
        UNKNOWN_enter();
    }

}

static void
ENQ_robot_timeout(uchar id)
{
    if (timerID == id)
        IDLE_enter();
}

static void
EOT_host_enter()
{
    state = EOT_host;
    timerID = MST_create(T2_ms, EOT_host_timeout);
}

static void
EOT_host_proc(uchar src, uchar val)
{
    passThrough(src, val);
    MST_cancel(timerID);

    if (src == HOST && 10 <= val && val <= 254)
        MSG_robot_enter(val);
    else
        UNKNOWN_enter();
}

static void
EOT_host_timeout(uchar id)
{
    if (timerID == id)
        IDLE_enter();
```

Communcation controller for SECS interface

```
}

static void
MSG_robot_enter(uchar len)
{
    state = MSG_robot;

    charRemain = len + 2;  /* include checksum */

    timerID = MST_create(T1_ms, MSG_robot_timeout);

    SECSMSG_init();  /* start saving message for SECS processing */
}

static void
MSG_robot_proc(uchar src, uchar val)
{
    uchar abortFlag;

    MST_cancel(timerID);

    if (src == HOST)
    {
        --charRemain;
        SECSMSG_proc(val, charRemain, &abortFlag);  /* store, process SECS message
        if (charRemain == 0)
        {
            if (abortFlag)
            {
                SER_put(ROBOT, ~val);
                BADCHECK_robot_enter();
            }
            else
            {
                SER_put(ROBOT, val);
                CHECK_robot_enter();
            }
        }
        else
        {
            SER_put(ROBOT, val);
            timerID = MST_create(T1_ms, MSG_robot_timeout);
        }
    }
    else
    {
        SER_put(HOST, val);
        UNKNOWN_enter();
    }
}

static void
MSG_robot_timeout(uchar id)
{
    if (timerID == id)
        IDLE_enter();
}

static void
CHECK_robot_enter()
{
    state = CHECK_robot;
    timerID = MST_create(T2_ms, CHECK_robot_timeout);
}
```

Communcation controller for SECS interface

```
static void
CHECK_robot_proc(uchar src, uchar val)
{
    passThrough(src, val);
    MST_cancel(timerID);

    if (src == ROBOT && (val == ACK || val == NAK))
    {
        if (val == ACK)
            SECSMSG_ack();   /* indicate ACK to SECS message processing */
        IDLE_enter();
    }
    else
        UNKNOWN_enter();
}

static void
CHECK_robot_timeout(uchar id)
{
    if (timerID == id)
        IDLE_enter();
}

static void
BADCHECK_robot_enter()
{
    state = BADCHECK_robot;
    timerID = MST_create(T2_ms, BADCHECK_robot_timeout);
    SER_disableInterrupt(HOST);
}

static void
BADCHECK_robot_proc(uchar src, uchar val)
{
    SER_enableInterrupt(HOST);

    MST_cancel(timerID);

    if (src == ROBOT && val == NAK)
    {
        SECSMSG_ack();   /* indicate ACK to SECS message processing */
        SER_put(HOST, ACK);
        IDLE_enter();
    }
    else
    {
        passThrough(src, val);
        UNKNOWN_enter();
    }
}

static void
BADCHECK_robot_timeout(uchar id)
{
    if (timerID == id)
        IDLE_enter();
}

static void
ENQ_host_enter()
{
    state = ENQ_host;
    timerID = MST_create(T2_ms, ENQ_host_timeout);
```

Communcation controller for SECS interface

```
}

static void
ENQ_host_proc(uchar src, uchar val)
{
    passThrough(src, val);

    if (src == HOST)
    {
        if (val == EOT)
        {
            MST_cancel(timerID);
            EOT_robot_enter();
        }
        else
            /* wait for another character from robot */ ;
    }
    else
    {
        MST_cancel(timerID);
        UNKNOWN_enter();
    }
}

static void
ENQ_host_timeout(uchar id)
{
    if (timerID == id)
        IDLE_enter();
}

static void
EOT_robot_enter()
{
    state = EOT_robot;
    timerID = MST_create(T2_ms, EOT_robot_timeout);
}

static void
EOT_robot_proc(uchar src, uchar val)
{
    passThrough(src, val);
    MST_cancel(timerID);

    if (src == ROBOT && 10 <= val && val <= 254)
        MSG_host_enter(val);
    else
        UNKNOWN_enter();
}

static void
EOT_robot_timeout(uchar id)
{
    if (timerID == id)
        IDLE_enter();
}

static void
MSG_host_enter(uchar len)
{
    state = MSG_host;

    charRemain = len + 2;  /* include checksum */
```

Communcation controller for SECS int rface

```
    timerID = MST_create(T1_ms, MSG_host_timeout);

    SECSMSG_init();  /* start saving message for SECS processing */
}

static void
MSG_host_proc(uchar src, uchar val)
{
    uchar dum;

    passThrough(src, val);
    MST_cancel(timerID);

    if (src == ROBOT)
    {
        --charRemain;
        SECSMSG_proc(val, charRemain, &dum);  /* store, process SECS message */
        if (charRemain == 0)
            CHECK_host_enter();
        else
            timerID = MST_create(T1_ms, MSG_host_timeout);
    }
    else
        UNKNOWN_enter();
}

static void
MSG_host_timeout(uchar id)
{
    if (timerID == id)
        IDLE_enter();
}

static void
CHECK_host_enter()
{
    state = CHECK_host;
    timerID = MST_create(T2_ms, CHECK_host_timeout);
}

static void
CHECK_host_proc(uchar src, uchar val)
{
    passThrough(src, val);
    MST_cancel(timerID);

    if (src == HOST && (val == ACK || val == NAK))
    {
        if (val == ACK)
            SECSMSG_ack();   /* indicate ACK to SECS message processing */
        IDLE_enter();
    }
    else
        UNKNOWN_enter();
}

static void
CHECK_host_timeout(uchar id)
{
    if (timerID == id)
        IDLE_enter();
}

static void
```

Communcation controll r for SECS intertac

```
IDENQ_host_enter()
{
    if (msgPtr == NULL || tryCount++ > MaxRetry)
    {
        msgPtr = NULL;
        IDLE_enter();
    }

    state = IDENQ_host;
    SER_put(HOST, ENQ);
    timerID = MST_create(T2_ms, IDENQ_host_timeout);

    SER_disableInterrupt(ROBOT);
}

static void
IDENQ_host_proc(uchar src, uchar val)
{
    if (src == HOST)
    {
        if (val == EOT)
        {
            MST_cancel(timerID);
            IDMSG_host_enter();
        }
        else
            /* wait for another character from robot */ ;
    }
}

static void
IDENQ_host_timeout(uchar id)
{
    if (timerID == id)
    {
        SER_enableInterrupt(ROBOT);
        IDENQ_host_enter();
    }
}

static void
IDMSG_host_enter()
{
    unsigned short checksum = 0;

    state = IDMSG_host;

    SER_put(HOST, msgLen);

    charRemain = msgLen;
    while (charRemain-- != 0)
    {
        checksum += *msgPtr;
        SER_put(HOST, *msgPtr++);
    }

    SER_put(HOST, (checksum & 0xff00) >> 8);
    SER_put(HOST, (checksum & 0x00ff));

    timerID = MST_create(T2_ms, IDMSG_host_timeout);
}

static void
IDMSG_host_proc(uchar src, uchar val)
```

Communcation controller for SECS interface

```
{
    if (src == HOST)
    {
        MST_cancel(timerID);

        SER_enableInterrupt(ROBOT);

        if (val == ACK)
        {
            msgPtr = NULL;
            IDLE_enter();
        }
        else
            IDENO_host_enter();
    }
}

static void
IDMSG_host_timeout(uchar id)
{
    if (timerID == id)
    {
        SER_enableInterrupt(ROBOT);
        IDENO_host_enter();
    }
}

static void
SECSMSG_init(void)
{
    bufIndex = 0;
}

static void
SECSMSG_proc(uchar inChar, uchar remain, uchar *abortFlag)
{
    if (bufIndex < BUF_SIZE)
        buf[bufIndex++] = inChar;

    if (remain == 0)
    {
        if ((buf[2] & 0x7f) == 3 && buf[3] == 1)    /* S3F1 */
            S3F1_proc(bufIndex, buf, abortFlag);
    }
}

static void
SECSMSG_ack(void)
{
    if ((buf[2] & 0x7f) == 3 && buf[3] == 1)        /* S3F1 */
        S3F1_complete();
}
```

S3F1 Message Processing

Processing an S3F1 message consists of validating the request, aborting the SECS message being transmitted to the robot, saving information from the incoming SECS request, initiating an ID system read, and formatting the SECS reply when the read completes. The three routines that do this are, in order:

S3F1_proc
: Validates the request (mainly checks the checksum) and, if it's OK, it aborts the SECS message being transmitted to the robot by calling SECS_abortMsgToRobot and stores away the device ID and system bytes from the incoming message. These are stored in the reply message buffer where they will be used in the reply. Note that the message must be checked and the message aborted before the second checksum byte is sent to the robot -- see the discussion in the SECS communication control.

S3F1_complete
: This routine initiates the ID system read. It is done here, and not in S3F1_proc, because if the robot fails to respond, we want to propogate a NAK to the host and not initiate a read. Starting it here just makes it easier to keep track of what is going on.

callback
: This routine formats the reply from the ID system. The reply message buffer is largely pre-formatted, since, except for the data itself, it does not change from read to read. A read error is indicated by returning a zero-length list of materials. A reader failure results in an S3F0 (Abort) reply.

S3F1.h

```
/*
  S3F1 message handling
  */

#ifndef _S3F1_H_
#define _S3F1_H_

extern void S3F1_proc(unsigned char length, unsigned char *msgPtr, unsigned char *a
extern void S3F1_complete(void);

#endif /* _S3F1_H_ */
```

S3F1.c

```
#include "exec.h"
#include "SECS.h"
#include "IDSys.h"

#include "S3F1.h"

/* forward declarations */
static void callback(uchar status, uchar dataLength, uchar *dataPtr);
```

S3F1 Message Processing

```
/* SECS format codes */
#define LIST   (000 << 2)
#define BINARY (010 << 2)
#define UINT1  (051 << 2)

#define FUNCTION_LOC         3   /* index of stream byte in SECS header */
#define SECS_HEADER_LENGTH  10
#define LIST_LENGTH_LOC     11   /* index of error zero-length byte */
#define ERR_MSG_SIZE        12   /* length of error message */
#define DATA_LENGTH_LOC     26   /* index of data length byte */
#define DATA_LOC            27   /* index of first data byte */
#define MAX_DATA_LENGTH      8   /* max number of data bytes */

#define BUF_SIZE     (DATA_LOC + MAX_DATA_LENGTH)
static uchar buf[BUF_SIZE] =     /* pre-loaded with S3F2 format */
{
    0, 0,               /* DevID */
    3, 2,               /* Stream/Function */
    0x80, 1,            /* Block count */
    0, 0, 0, 0,         /* System bytes */
    LIST | 1, 2,        /* L, 2 */
    BINARY | 1, 1,      /* <MF> Material format code, 1 byte */
    2,                      /* 2 => cassette */
    LIST | 1, 1,        /* L, m m = # cassettes = 1 */
    LIST | 1, 3,        /* L, 3 */
    BINARY | 1, 1,      /* <LOC> Location code, 1 byte */
    0,                      /* 0 -- our own location code? */
    UINT1 | 1, 1,       /* <QUA> Quantitiy in loc, 1 byte */
    1,                      /* 1 -- always 1 cassette */
    BINARY | 1, 8,      /* <MID> Material ID, 8 bytes */
    0                   /* start of data */
};

void
S3F1_proc(uchar length, uchar *msgPtr, uchar *abortFlag)
{
    uchar i;
    unsigned short checksum;

    *abortFlag = FALSE;

    if (   length = SECS_HEADER_LENGTH + 2      /* including checksum */
        && (msgPtr[0] & 0x80) == 0)      /* R-bit == 0 => msg to robot */
    {
        checksum = 0;
        for (i = 0; i < SECS_HEADER_LENGTH; i++)
            checksum += msgPtr[i];

        if (   (checksum & 0xff00) >> 8 != msgPtr[SECS_HEADER_LENGTH]
            || (checksum & 0x00ff)      != msgPtr[SECS_HEADER_LENGTH + 1])
            return;

        buf[0] = msgPtr[0] | 0x80;       /* copy DevID; set R-bit */
        buf[1] = msgPtr[1];
        buf[6] = msgPtr[6];              /* copy system bytes */
        buf[7] = msgPtr[7];
        buf[8] = msgPtr[8];
        buf[9] = msgPtr[9];

        *abortFlag = TRUE;     /* cause SECS comm control to abort message being sen
    }
}

void
```

S3F1 Message Processing

```
S3F1_complete(void)
{
    ID_read(callback);
}

static void
callback(uchar status, uchar dataLength, uchar *dataPtr)
{
    uchar i;
    uchar msgLen;

    switch (status)
    {
    case SUCCESS:
        buf[FUNCTION_LOC] = 2;
        buf[LIST_LENGTH_LOC] = 2;

        if (dataLength > MAX_DATA_LENGTH)
            dataLength = MAX_DATA_LENGTH;

        buf[DATA_LENGTH_LOC] = dataLength;
        for (i = 0; i < dataLength; i++)
            buf[DATA_LOC + i] = dataPtr[i];

        msgLen = DATA_LOC + dataLength;
        break;

    case READ_FAILED:
        buf[FUNCTION_LOC] = 2;
        buf[LIST_LENGTH_LOC] = 0;    /* Zero-length list => no read */
        msgLen = ERR_MSG_SIZE;
        break;

    case READER_FAILED:
    default:
       buf[FUNCTION_LOC] = 0;        /* S3F0 => reader failed */
       msgLen = SECS_HEADER_LENGTH;
    }

    SECS_sendMsgToHost(msgLen, buf);
}
```

Communication Control for ASCII Interface

The module is responsible for the coordination of all communication with external devices -- the host, the robot, the ID system -- for the ASCII message protocol configuration.

ASCII.h

```
/*
  Communication control for ASCII interface
  */

#ifndef _ASCII_H_
#define _ASCII_H_

extern void ASCII_init(void);

#endif /* _ASCII_H_ */
```

ASCII.c

```
#include <stdio.h>
#include <string.h>

#include "exec.h"
#include "wd.h"
#include "msTimer.h"
#include "serialPorts.h"
#include "IDSys.h"

#include "ASCII.h"

#define READ_COMMAND  "READID"

/* Convenient ASCII definitions */
#define CR 0x0d
#define LF 0x0a
```

Communication Control for ASCII Interface

```
/* forward declarations */
static void receiveHost(uchar inChar);
static void callback(
    unsigned char status,
    unsigned char dataLength,
    unsigned char *dataPtr);
static void sendRobot(uchar dum);
static void receiveRobot(uchar inChar);
static void sendHostMsg(unsigned char length);
static void sendHost(uchar dum);
static void timeout(uchar dum);

/* globals */
enum
{
    RECEIVING_FROM_HOST,
    SENDING_TO_ROBOT,
    RECEIVING_FROM_ROBOT,
    SENDING_TO_HOST,
    READING_ID
};
static uchar state;

#define BUF_SIZE    1000
static xdata uchar buf[BUF_SIZE + 1];

static uchar bufIndex;
static uchar charRemain;

void
ASCII_init(void)
{
    SER_setHandler(HOST, RECEIVE, receiveHost);
    SER_setHandler(HOST, SEND, sendHost);
    SER_setHandler(ROBOT, RECEIVE, receiveRobot);
    SER_setHandler(ROBOT, SEND, sendRobot);

    state = RECEIVING_FROM_HOST;
    bufIndex = 0;

    timeout(0);
}

static void
receiveHost(uchar inChar)
{
    if (state != RECEIVING_FROM_HOST)
        return;

    buf[bufIndex] = inChar;

    if (bufIndex < BUF_SIZE)
        bufIndex++;

    if (inChar == LF || inChar == CR)
    {
        if (   strlen(READ_COMMAND) == bufIndex - 1
            && strncmp((char *) buf, READ_COMMAND, bufIndex - 1) == 0)
        {
            state = READING_ID;
            ID_read(callback);
        }
        else
```

Communication Control for ASCII Interface

```
        {
            state = SENDING_TO_ROBOT;
            charRemain = bufIndex;
            bufIndex = 0;
            sendRobot(0);
        }
    }
}

static void
callback(
    unsigned char status,
    unsigned char dataLength,
    unsigned char *dataPtr)
{
    int i;

    if (state != READING_ID)
        return;

    if (status == SUCCESS)
    {
        sprintf((char *) buf, "LPT> %d ", status);

        for (i = 0; i < dataLength; i++)
        {
            sprintf((char *) &buf[strlen((char *) buf)], "%d ", dataPtr[i]);
        }

        sprintf((char *) &buf[strlen((char *) buf)], "LPT>\n");
    }
    else
        sprintf((char *) buf, "LPT> %d LPT>\n", status);

    sendHostMsg(strlen((char *) buf));
}

static void
sendRobot(uchar dum)
{
    if (state != SENDING_TO_ROBOT)
        return;

    if (charRemain-- > 0)
    {
        SER_put(ROBOT, buf[bufIndex++]);
    }
    else
    {
        state = RECEIVING_FROM_ROBOT;
        bufIndex = 0;
    }
}

static void
receiveRobot(uchar inChar)
{
    if (state != RECEIVING_FROM_ROBOT)
        return;

    buf[bufIndex] = inChar;

    if (bufIndex < BUF_SIZE)
        bufIndex++;
```

Communication Control for ASCII Interface

```
    if (inChar == LF || inChar == CR)
    {
        state = RECEIVING_FROM_ROBOT;
        sendHostMsg(bufIndex);
    }
}

static void
sendHostMsg(unsigned char length)
{
    state = SENDING_TO_HOST;
    charRemain = length;
    bufIndex = 0;
    sendHost(0);
}

static void
sendHost(uchar dum)
{
    if (state != SENDING_TO_HOST)
        return;

    if (charRemain-- > 0)
    {
        SER_put(HOST, buf[bufIndex++]);
    }
    else
    {
        WD_reset();
        state = RECEIVING_FROM_HOST;
        bufIndex = 0;
    }
}

static void
timeout(uchar dum)
{
    if (state == RECEIVING_FROM_HOST && bufIndex == 0)
        WD_reset();

    MST_create(10000, timeout);
}
```

RF ID Interface

This module initiates an RF system operation to read the transponder in the SMIF pod currently placed on the robot, and it delivers the data it reads to a callback routine. The only service it exports to the rest of the IMM is ID_read, and the only parameter ID_read takes is the callback routine. The status of the call, length of data, and a pointer to the data read are passed to the callback. The TIRIS system responds with an error message if the read fails for some reason (no transponder, multiple transponders, corrupted message, etc.), so the status differentiates between a failed read attempt (error message) and the reader failing (no response). The eight data bytes are sent to the callback routine least significant byte first.

Internally, the module goes through a very simple sequence of states: it is IDLE, an ID_read call makes it busy SENDING_COMMAND, after completing the send it waits while RECEIVING_REPLY, and after receiving the reply (or timing out, if the reader should fail) and sending the data to the callback routine, it becomes IDLE again. The read command is just a pre-loaded five-byte array of characters stored in readCmd. The response is stored in readReply. The receiving routine checks the length, status, and BCC (longitudinal redundancy check byte, the Xor of all the bytes in the message excluding the start mark) in the response. The format of the TIRIS messages are documented in the TIRIS micro-reader reference manual, available from Texas Instruments.

IDSys.h

```
/*
  ID system interface

  Reads from RFID system, invokes callback on completion
  */

#ifndef _IDSYS_H_
#define _IDSYS_H_

/* Read status */
enum
{
    SUCCESS,
    READ_FAILED,
    READER_FAILED
};

typedef void (*ID_callback)(
    unsigned char status,
    unsigned char dataLength,
    unsigned char *dataPtr);

extern unsigned char ID_read(ID_callback cb);

#endif /* _IDSYS_H_ */
```

IDSys.c

RF ID Interface

```
#include "reg80c320.h"
#include "exec.h"
#include "IDPort.h"
#include "msTimer.h"

#include "IDSys.h"

/* ID reader timeout, milliseconds */
#define ID_TIMEOUT     200

/* Convenient ASCII definitions */
#define SOH   0x01

/* forward declarations */
static void sendID(uchar dum);
static void receiveID(uchar inChar);
static void reply(uchar status);
static void timeout(uchar timerID);

/* globals */
enum                /* allowed states */
{
    IDLE,
    SENDING_COMMAND,
    RECEIVING_REPLY
};
static uchar state = IDLE;

static uchar charRemain;
static uchar bufIndex;
static uchar readCmd[] = { SOH, 0x02, 0x08, 0x32, 0x38 };

static uchar timerID;

static ID_callback callback;

#define SUCCESS_REPLY_LEN  12
#define READ_REPLY_LEN     SUCCESS_REPLY_LEN
static uchar readReply[READ_REPLY_LEN];

unsigned char
ID_read(ID_callback cb)
{
    if (state != IDLE)
        return FALSE;       /* already busy */

    if (cb == NULL)
        return FALSE;       /* need to do something with result */

    ID_setReceiveHandler(receiveID);
    ID_setSendHandler(sendID);

    state = SENDING_COMMAND;
    timerID = MST_create(ID_TIMEOUT, timeout);
    bufIndex = 0;
    charRemain = sizeof(readCmd) / sizeof(readCmd[0]);
    callback = cb;
    sendID(0);

    return TRUE;
}

static void
```

RF ID Interface

```
sendID(uchar dum)
{
    if (state != SENDING_COMMAND)
        return;

    if (charRemain-- > 0)
    {
        ID_put(readCmd[bufIndex++]);
    }
    else
    {
        state = RECEIVING_REPLY;
        bufIndex = 0;
        charRemain = 2;
    }
}

static void
receiveID(uchar inChar)
{
    if (state != RECEIVING_REPLY)
        return;

    /* first character MUST be an SOH start mark */
    if (bufIndex == 0 && inChar != SOH)
        return;

    if (bufIndex < READ_REPLY_LEN)
        readReply[bufIndex] = inChar;

    bufIndex++;

    if (bufIndex == 2)          /* just read header */
    {
        charRemain = readReply[1] + 1;  /* include BCC */
    }
    else if (--charRemain == 0)  /* finished message */
    {
        state = IDLE;

        MST_cancel(timerID);

        if (bufIndex == SUCCESS_REPLY_LEN)
        {
            uchar i, bcc;

            /* Check status */
            if (readReply[2] != 0x0C)
                reply(READER_FAILED);

            /* check BCC -- X'OR bytes of message (skip start mark) */
            bcc = 0;
            for (i = 1; i < SUCCESS_REPLY_LEN - 1; i++)
                bcc ^= readReply[i];

            if (bcc == readReply[SUCCESS_REPLY_LEN - 1])
                reply(SUCCESS);
            else
                reply(READER_FAILED);
        }
        else
        {
            reply(READ_FAILED);
        }
```

RF ID Interface

```
    }
}

static void
reply(uchar status)
{
    if (status == SUCCESS)
        callback(SUCCESS, 8, &readReply[3]);
    else
        callback(status, 0, NULL);

    callback = NULL;
}

static void
timeout(uchar dum)
{
    state = IDLE;

    reply(READER_FAILED);
}
```

ID Port Handler

The ID port handler is configured very similarly to the serial port handlers, except that running the external DUART connection are somewhat different from running the internal serial ports. The ID port handler services interrupts from the external DUART and invokes a callback routine, if needed; and it sends a character to the DUART. The services are the same, without the interrupt enable/disable capability:

ID_init
: Initializes the ID port handler, which handles interrupts from EXT4, which is connected to the DUART. The DUART has its own crystal, so no internal timer is used. Its registers are mapped to memory locations in external data.

ID_put
: Initiates sending a character to the ID port, but does not wait for completion. If the port is currently busy sending a character, it spin waits for the transmission to complete, though this does not normally occur in the current system.

ID_setSendHandler, ID_setReceiveHandler
: Sets the ID port SEND and RECEIVE event interrupt handlers.

IDPort.h

```
/*
  Initializes, handles interrupts for
    external DUART port 0 (on ext4, interrupt 10)
  */

#ifndef _IDPORT_H_
#define _IDPORT_H_

extern void ID_init(void);

extern void ID_setSendHandler(
    void (*handler)(unsigned char dum));    /* NULL => ignore interrupt */
extern void ID_setReceiveHandler(
    void (*handler)(unsigned char value)); /* NULL => ignore interrupt */

extern void ID_put(unsigned char value);

#endif /* _IDPORT_H_ */
```

IDPort.c

```
/*
  ID port handler
  */

#include "reg80c320.h"
#include "exec.h"

#include "IDPort.h"
```

ID Port Handler

```
/*
  DUART register definitions -- National PC16552D
  */

#define ID_BASE(n) 0xF800 + n * 0x100 xdata

at ID_BASE(0) uchar RBR;     /* read only, DLAB = 0 */
at ID_BASE(0) uchar THR;     /* write only, DLAB = 0 */
at ID_BASE(1) uchar IER;     /* DLAB = 0 */
at ID_BASE(2) uchar IIR;     /* read only */
at ID_BASE(2) uchar FCR;     /* write only */
at ID_BASE(3) uchar LCR;
at ID_BASE(4) uchar MCR;

at ID_BASE(5) uchar LSR;

at ID_BASE(7) uchar SCR;     /* scratch reg, for debugging */

at ID_BASE(0) uchar DLL;     /* DLAB = 1 */
at ID_BASE(1) uchar DLM;     /* DLAB = 1 */

/* Interrupt handlers */
static void (*sendHandler)(uchar dum);
static void (*receiveHandler)(uchar value);

/* Software output busy flag */
static bool outBusy;

/* forward declarations */
static void null_handler(unsigned char dum);

void
ID_init(void)
{
    outBusy = FALSE;

    ID_setSendHandler(NULL);
    ID_setReceiveHandler(NULL);

    /* Set up IDSYS interface to DUART on external int 4 */
    LCR = 0x80;     /* DLAB = 1 */
    DLM = 0;        /* 9600 baud, 18.432 MHz crystal */
    DLL = 120;

    LCR = 0x03;     /* DLAB = 0; 8-N-1 */
    FCR = 0;        /* disable FIFOs */
    IER = 0x03;     /* enable data avail, xmit done interrupts */
    MCR = 0;        /* clear modem control, just in case */
    EXIF &= ~0x40; /* clear interrupt 4 flag, in case it's set */
    EX4 = 1;        /* enable external interrupt 4 */
}

static void
null_handler(unsigned char dum)
{
    return;  /* do-nothing handler, convenient default */
}

static void
handle_IDSYS(void) interrupt 10
{
    uchar src;
    uchar inChar;
```

ID Port Handler

```
    EXIF &= ~0x40;      /* clear interrupt */

    src = IIR;

    switch (src)
    {
    case 1:
        break;          /* no interrupt (??), do nothing */

    case 2:
        outBusy = FALSE;
        (sendHandler)(0);
        break;

    case 4:
        inChar = RBR;  /* read character from DUART register */
        (receiveHandler)(inChar);
        break;

    default:
        panic(UNKNOWN_INTERRUPT);
    };
}

void
ID_setSendHandler(void (*new_handler)(uchar dum))
{
    if (new_handler == NULL)
        sendHandler = null_handler;
    else
        sendHandler = new_handler;
}

void
ID_setReceiveHandler(void (*new_handler)(uchar value))
{
    if (new_handler == NULL)
        receiveHandler = null_handler;
    else
        receiveHandler = new_handler;
}

void
ID_put(uchar outChar)
{
    if (outBusy)
    {
        while (LSR & 0x20 == 0)    /* wait for last send to complete */
            continue;
    }

    outBusy = TRUE;
    THR = outChar;
}
```

Serial Port Handlers

The serial port interrupt handlers perform two basic functions: they set up and service built-in serial port interrupts, invoking application callback routines, if needed; and they send characters out the serial ports. The services offered to the rest of the system include:

SER_init
: Initializes the serial port handlers and timer1, which is used to generate the baud rate. Timer1 is set to run in mode 2, 8-bit auto-reload mode, at 9600 baud (on a 33 MHz machine). Send and receive interrupts are enabled for both PORT0 and PORT1, output channels are marked idle, and a dummy callback is registered for both SEND and RECEIVE events.

SER_put
: Initiates sending a character on the specified port, but does not wait for completion. If the port is currently busy sending a character, it spin waits for the transmission to complete. SER_put determines that output is currently in progress by inspecting the outBusy flag for that channel. This flag is set by SER_put, and cleared by the interrupt service routine.

  No buffering or queueing service is provided by SER_put. This means that if it is called from an interrupt service routine, no interrupts will be serviced during the call, which could take as long as a millisecond (one character at 9600 baud). This has several consequences in the current IMM system. For one, it means that the robot and host interfaces should be operated at the same baud rate, since otherwise one could overrun the other in the absence of speed matching buffering. For another, it means that if a routine has more than one character to send, it must either supply its own output interrupt service routine (as ID System does), or be used in such a situation that missing other interrupts can be tolerated (as is the case when the SECS communication controller sends an IMM message to the host).

SER_setHandler
: Associates a callback routine with a particular port and event (SEND or RECEIVE). If the callback routine is NULL, then a do-nothing dummy callback routine is registered.

SER_enableInterrupt, SER_disableInterrupt
: Enable/disable interrupts on the indicated port. These routines are used by the SECS communication control to shut down and restore communication with the robot during IMM-host conversations.

The actual interrupt handlers, handle_port0 and handle_port1, do little more than reset the hardware and software flags associated with the port, and invoke the registered callback routine. Note the peculiar interface to the port1 handler: this is a result of working around a problem with the Franklin DS80C320 simulator handling of port1 interrupts. See the workaround discussion in the main exec routine for more detail.

---

serialPorts.h

```
/*
  Initializes, handles interrupts for
    internal serial port 0 (interrupt 4)
    internal serial port 1 (interrupt 7)

  The serial port driver "owns" timer1 (interrupt 3),
```

Serial Port Handlers

```
  which is used to generate the baud rate
  */

#ifndef _SERIALPORTS_H_
#define _SERIALPORTS_H_

/* assign ROBOT, HOST to PORTs */
#define ROBOT    PORT0
#define HOST     PORT1

/* port identifiers */
enum
{
    PORT0 = 0,
    PORT1,
    NUM_PORTS
};

/* events */
enum
{
    SEND = 0,
    RECEIVE,
    NUM_EVENTS
};

extern void SER_init(void);

extern void SER_enableInterrupt(unsigned char port);
extern void SER_disableInterrupt(unsigned char port);

extern void SER_setHandler(
    unsigned char port,
    unsigned char event,
    void (*handler)(unsigned char value)); /* NULL => ignore interrupt */

extern void SER_put(unsigned char port, unsigned char value);

#endif /* _SERIALPORTS_H_ */
```

---

serialPorts.c

```
/*
  Serial port handlers
  */

#include "reg80c320.h"
#include "exec.h"

#include "serialPorts.h"

/* Array of interrupt handlers */
typedef void (*t_handler)(unsigned char value);
static t_handler handler[NUM_PORTS][NUM_EVENTS];

/* Vector of software output busy flags */
static bool outBusy[NUM_PORTS];

/* forward declarations */
static void null_handler(unsigned char dum);
```

S nal Port Handl rs

```
void
SER_init(void)
{
    int i, j;

    for (i = 0; i < NUM_PORTS; i++)
    {
        outBusy[i] = FALSE;
        for (j = 0; j < NUM_EVENTS; j++)
            handler[i][j] = null_handler;
    }

    /* Set up timer 1 to run 9600 baud serial I/O */
    TMOD = (TMOD & 0x0f) | 0x20; /* Timer 1 => mode 2 (8-bit reload) */
    TH1  = 0xf7;    /* Reload value for 33.00 MHz, 9600/19200 baud */
    TL1  = 0xf7;    /* Make first time-out correct */
    TR1  = 1;       /* Start timer */

    /* Set up internal serial port 0 */
    PCON &= ~0x80; /* set SMOD for 9600 baud */
    SCON0 = 0x50;  /* mode 1 (vari baud), receive enable */
    SER_enableInterrupt(PORT0);

    /* Set up internal serial port 1 */
    SMOD_1 = 0;     /* set SMOD for 9600 baud */
    SCON1  = 0x50; /* mode 1 (vari baud), receive enable */
    SER_enableInterrupt(PORT1);
}

void
SER_enableInterrupt(unsigned char port)
{
    if (port == PORT0)
    {
        RI_0  = 0;      /* clear input interrupt */
        TI_0  = 0;      /* clear output interrupt */
        ES0 = 1;        /* enable port 0 interrupt */
    }
    else if (port == PORT1)
    {
        RI_1   = 0;     /* clear input interrupt */
        TI_1   = 0;     /* clear output interrupt */
        ES1 = 1;        /* enable port 1 interrupt */
    }
}

void
SER_disableInterrupt(unsigned char port)
{
    if (port == PORT0)
        ES0 = 0;
    else if (port == PORT1)
        ES1 = 0;
}

static void
null_handler(unsigned char dum)
{
    return;  /* do-nothing handler, convenient default */
}

static void
handle_port0(void) interrupt 4
{
```

Serial Port Handlers

```
    uchar inChar;

    if (TI_0 == 1)
    {
        TI_0 = 0;          /* clear output interrupt */
        outBusy[PORT0] = FALSE;
        (handler[PORT0][SEND])(0);
    }

    if (RI_0 == 1)
    {
        inChar = SBUF0;    /* read character from port0 buffer */
        RI_0 = 0;          /* clear port0 receiver interrupt flag */
        (handler[PORT0][RECEIVE])(inChar);
    }
}

#ifdef SIM
void
handle_port1(void)
#else
static void
handle_port1(void) interrupt 7
#endif
{
    uchar inChar;

    if (TI_1 == 1)
    {
        TI_1 = 0;          /* clear output interrupt */
        outBusy[PORT1] = FALSE;
        (handler[PORT1][SEND])(0);
    }

    if (RI_1 == 1)
    {
        RI_1 = 0;          /* clear port1 receiver interrupt flag */
        inChar = SBUF1;    /* read character from port1 buffer */
        (handler[PORT1][RECEIVE])(inChar);
    }
}

void
SER_setHandler(
    unsigned char port,
    unsigned char direction,
    void (*new_handler)(uchar value))
{
    if (new_handler == NULL)
        handler[port][direction] = null_handler;
    else
        handler[port][direction] = new_handler;
}

void
SER_put(uchar port, uchar outChar)
{
    switch (port)
    {
    case PORT0:
        if (outBusy[PORT0])
        {
            while (TI_0 == 0)
                continue;     /* wait for previous output to complete */
```

Serial Port Handlers

```
            TI_0 = 0;
        }
        SBUF0 = outChar;
        break;

    case PORT1:
        if (outBusy[PORT1])
        {
            while (TI_1 == 0)
                continue;     /* wait for previous output to complete */
            TI_1 = 0;
        }
        SBUF1 = outChar;
        break;

    default:
        return;
    }

    outBusy[port] = TRUE;
}
```

Millisecond Timer Handler

This module implements in software a general-purpose millisecond-level timer facility, which allows a client to register a routine to be run a designated number of milliseconds in the future. This is used to implement timers in the communication protocol, and to provide a safety net should the RF ID system fail to respond. It uses the built-in timer0.

Timers are created by invoking MST_create, passing it the delay in milliseconds and a pointer to the callback routine. The return value is a timerID, a value that may be passed to MST_cancel to cancel the timer. The callback routine takes one argument, the timerID for which it was invoked, and should return no value. (Taking a second argument, such as a byte of user-defined data, was considered, but the IMM application did not have any use for it, and it would have increased the size of the timer data structure.)

The basic data structure that the timer routines use simply contains *ticksRemain*, the number of ticks left until the timer expires, and the callback routine address. The timerID is an index into a statically allocated array of these structures. Every time the interrupt handler for timer0 is entered, the number of ticks remaining in each timer struct is checked, and when it drops to zero the callback routine is invoked. MST_create just searches the array for an unused entry (if the *ticksRemain* is zero, the timer isn't being used), and MST_cancel just sets *ticksRemain* to zero. The initialization routine, MST_init, clears the array and starts up timer0.

One limitation to the use of millisecond timers arises from the fact that the current implementation maintains a static array of NUM_TIMERS timer data structures in the on-chip data area, so there is an upper limit on the number of simultaneously available timers. NUM_TIMERS is currently set to four.

Another limitation arises from the fact that in the timer data structure the number of ticks left before expiration is maintained as a two-byte integer. Using a 10 ms. tick (27500 cycles of the 33.00 MHz crystal, counted every 12 cycles), this allows a maximum period of 655.36 sec. (about 10 min.) with a resolution of 10 ms.

Note that in order to get a reasonably long tick we use timer0 in mode 1, 16-bit mode, so we have to reload the count manually in the interrupt handler. This behavior is somewhat different from the way timer1 is used in auto-reload mode to clock the serial ports.

---

msTimer.h

```
/*
  Millisecond software timers

  Uses timer0 (interrupt 1)

  This implementation has a maximum wait of 655.36 sec (about 10 min.)
  with a resolution of 10 ms.

  This implementation currently only allows NUM_TIMERS simultaneous
  timers.
  */
```

Millisecond Timer Handler

```
#ifndef _MSTIMER_H_
#define _MSTIMER_H_

#define LOG_NUM_TIMERS     2
#define NUM_TIMERS         (1 << LOG_NUM_TIMERS)
#define INVALID_TIMERID    NUM_TIMERS

extern void MST_init(void);

extern unsigned char MST_create(
    unsigned long ms,
    void (*callback)(unsigned char timerID));

extern void MST_cancel(unsigned char timerID);

#endif /* _MSTIMER_H_ */
```

---

msTimer.c

```
/*
  Use timer0 to implement software timers

  27500 cycles of the 33.00 MHz crystal, counted every 12 cycles,
  takes 10 ms.

  We use mode 1, the 16-bit timer mode, and reload manually every interrupt.

  Using a two-byte tick count field and a 10 ms tick we get a
  maximum timer value of 655.36 sec, about 10 min.
  */


#include "reg80c320.h"
#include "exec.h"

#include "msTimer.h"

#define MS_RESOLUTION 10
#define NUM_CYCLES    27500
#define RELOAD_VALUE  65536 - NUM_CYCLES
#define RELOAD_HI     (RELOAD_VALUE & 0xff00) >> 8
#define RELOAD_LO     RELOAD_VALUE & 0x00ff

static data struct
{
    unsigned short ticksRemain;  /* 0 => not in use */
    uchar timerID;
    void (*callback)(uchar timerID);
} timers[NUM_TIMERS];

static uchar currTimerBase;

void
MST_init(void)
{
    uchar i;

    currTimerBase = 0;

    /* clear soft timer data structure */
    for (i = 0; i < NUM_TIMERS; i++)
```

Millisecond Timer Handler

```
        timers[i].ticksRemain = 0;

    /* set up timer0 */
    TMOD = (TMOD & 0xf0) | 0x01; /* Timer 0 => mode 1 (16-bit timer) */
    TH0  = RELOAD_HI;
    TL0  = RELOAD_LO;
    TR0  = 1;        /* Start timer */
    ET0  = 1;        /* Enable timer0 interrupts */
}

static void
handleMsTimer(void) interrupt 1
{
    int i;

    TH0 = RELOAD_HI;
    TL0 = RELOAD_LO;
    TR0 = 1;

    for (i = 0; i < NUM_TIMERS; i++)
    {
        if (timers[i].ticksRemain != 0)
        {
            if (--timers[i].ticksRemain == 0)
            {
                (timers[i].callback)(timers[i].timerID);
            }
        }
    }
}

uchar
MST_create(
    unsigned long ms,
    void (*callback)(uchar timerID))
{
    uchar i;
    uchar timerID;

    ms = (ms < MS_RESOLUTION) ? MS_RESOLUTION : ms;

    for (i = 0; i < NUM_TIMERS; i++)
    {
        if (timers[i].ticksRemain == 0)
        {
            timerID = (currTimerBase++ << LOG_NUM_TIMERS) + i;
            timers[i].ticksRemain = (unsigned short) (ms / MS_RESOLUTION);
            timers[i].timerID = timerID;
            timers[i].callback = callback;
            return timerID;
        }
    }

    panic(NO_MORE_TIMERS);
}

void
MST_cancel(uchar timerID)
{
    uchar i;

    i = timerID & (NUM_TIMERS - 1);

    if (timers[i].timerID == timerID)
```

Millisecond Timer Handler

```
        timers[i].ticksRemain = 0;
}
```

Watchdog Timer Handler

Watchdog Timer Handler

This handler implements a simple watchdog timer, based on the DS80C320's hardware capabilities. It's purpose is to reset the micro-controller in case it get hung for some reason. "Getting hung" is inferred from a lack of change when change is expected. The timer is initialize by the executive during initialization, and reset periodically by the communication control finite state machines as external events move them through their sequence of state transitions. The timer will accumulate MAX_TICKS ticks before resetting the CPU. The current values, MAX_TICKS = 128 and 2^26 clocks @ 33 MHz per tick, allow 4-5 minutes to pass without progress before the watchdog goes off.

Note the use of the TIMED_ACCESS macro to set the protected registers associated with the watchdog timer.

---

wd.h

```
#ifndef _WD_H_
#define _WD_H_

extern void WD_init(void);

extern void WD_reset(void);

#endif /* _WD_H_ */
```

---

wd.c

```
#include "reg80c320.h"

#include "wd.h"

/* Number of ~2 sec TICKS before reset */
#define MAX_TICKS        128

static unsigned char resetCount;

void
WD_init(void)
{
    resetCount = 0;    /* clear software count */

    CKCON |= 0xC0;     /* set watchdog interval to 2 ^ 26 clocks */
                       /* (about 2 sec at 33 MHz) */

    TIMED_ACCESS(RWT = 1);    /* restart (clear) watchdog timer */

    TIMED_ACCESS(EWT = 1);    /* enable reset mode */

    TIMED_ACCESS(WDIF = 0);   /* clear watchdog interrupt flag */
    PWDI = 1;                 /* set watchdog interrupt priority to high */
    EWDI = 1;                 /* enable watchdog interrupt */
}

void
WD_reset(void)
```

Watchdog Timer Handler

```
{
    resetCount = 0;
}

static void
handle_WatchDog(void) interrupt 12
{
    if (resetCount++ < MAX_TICKS)
        TIMED_ACCESS(RWT = 1);

    TIMED_ACCESS(WDIF = 0);
}
```

Executive Routine

The main routine in the program serves only to initialize the various modules, then enable interrupts and enter a do-nothing idle loop (though note the conditionally compiled workaround).

The panic routine is a simple-minded way of stopping the IMM: it disables interrupts, sets an error code at the beginning external data memory, and spins. There might be something better to do -- add a display to the board to hold the error code? -- but I can't think of anything. I have never seen this routine called in any phase of our testing. The error codes are defined in exec.h.

There are a couple of conditional compilation parameters defined in the executive. Two of these, ASCII and SECS, control whether code for the ASCII or SECS interface is run during initialization. The other, SIM, controls the inclusion of code to work around the inability of the Franklin software to simulate serial port 1 interrupts.

---

exec.h

```
#ifndef _EXEC_H_
#define _EXEC_H_

/*
  Define when compiling to run on the Franklin DS80C320 simulator
  Comment out when compiling to run on actual micro-controller
#define SIM
*/

typedef unsigned char uchar;
typedef unsigned char bool;

#ifndef TRUE
#define TRUE (0 == 0)
#define FALSE (! TRUE)
#endif

#ifndef NULL
#define NULL 0
#endif

/*
  panic codes
  */
#define NO_MORE_TIMERS      1
#define UNKNOWN_INTERRUPT   2

extern void panic(unsigned char);

#endif /* _EXEC_H_ */
```

---

exec.c

```
/*
  Main initialization driver, entered on reset.
```

Executive Routine

```
  Add calls to initialization routines after "here -->"
  */

/*
  Define appropriate interface type
#define ASCII
*/
#define SECS

#include "reg80c320.h"

#include "wd.h"
#include "serialPorts.h"
#include "msTimer.h"
#include "ASCII.h"
#include "SECS.h"

#include "exec.h"

void
main(void)
{
    /*
      Disable interrupts, including external interrupts
      and power fail.  Initialization routines enable
      their own interrupts.
      */
    IE = 0;
    EIE = 0;
    EPFI = 0;

    /*
      Call initialization routines here -->
         (Note: order is important!!)
      */
    WD_init();     /* watchdog timer */
    MST_init();    /* millisecond timer */
    SER_init();    /* serial ports 0 & 1, external interrupt 4 */

#ifdef ASCII
    ASCII_init(); /* communication control for ASCII interface */
#endif
#ifdef SECS
    SECS_init();  /* communication control for SECS interface */
#endif

    EA = 1;    /* globally enable interrupts */

    for (;;)
    {

#ifdef SIM
        /*
          This is a workaround for a bug in the Franklin DS80C320 simulator.
          Since the simulator fails to generate interrupts properly for
          serial port 1, we just spin in the idle loop waiting for RI or TI
          to be set, and then we call the handler directly.  Note that
          we must also change the declaration of the handler, so it does
          an RET instead of a RETI.  This code is not used in production
          (though it's easy to forget to change it).
          */
```

Executive Routine

```
        if (RI_1 == 1 || TI_1 == 1)
        {
            EA = 0;
            handle_port1();
            EA = 1;
        }
#endif
        continue;           /* background task -- idle loop */
    }
}

/*
  Panic routine puts an error code at start of external data,
  and spins to halt processor

  There must be something better to do.
  */

at 0x0000 xdata uchar HaltCode = 0;

void
panic(uchar errCode)
{
    EA = 0;                 /* turn off interrupts */

    HaltCode = errCode;     /* set error code */

halt:
    goto halt;              /* spin */
}
```

This invention has been described and illustrated in connection with certain preferred embodiments which are illustrative of the principles of the invention. However, it should be understood that various modifications and changes may readily occur to those skilled in the art, and it is not intended to limit the invention to the construction and operation of the embodiment shown and described herein. Accordingly, additional modifications and equivalents may be considered as falling within the scope of the invention as defined by the claims hereinbelow.

WHAT IS CLAIMED IS:

1. A module for use in a system for processing articles, which system includes a plurality of machine tools for processing articles, a pod for carrying the articles to be processed by said machine tools from one machine tool to another, a host processing controller associated with said machine tools for controlling the operation thereof, a robot connected to each said machine tool for receiving said pod, opening said pod and for transporting the articles from within said pod into position on said machine tool for processing, and an identification means carried by said pod for identifying a particular pod and the articles carried therein, said module comprising single wire connection between said identification means, said host controller and said robot.

2. The module according to claim 1 wherein said single wire connection is carried by a microprocessor having means for identifying the source of a communication signal for routing said communication signals between said identification means and said host controller and between said host controller and said robot depending upon its source.

3. The module according to claim 2 further comprising a reader connected to said microprocessor for reading said identification means when said pod is placed on said machine tool.

4. The module according to claim 3 wherein said identification means has an RF transmitter and said reader comprises an RF receiver.

5. The module according to claim 3 wherein said identification means has an infrared transmitter and said reader comprises an infrared receiver.

6. The module according to claim 3 wherein said identification means is a barcode and said reader comprises an optical character reader.

7. The module according to claim 3 further comprising detecting means mounted on said machine tool for detecting the presence of said pod, said detecting means connected with said host controller, whereby when said host controller receives a message from said detecting means that a pod is in place for processing on said machine tool said host controller will issue a command to said reader to identify said pod by reading its identification means.

8. The module according to claim 7 further comprising a power source and power regulation means for providing power to said microprocessor and to said reader.

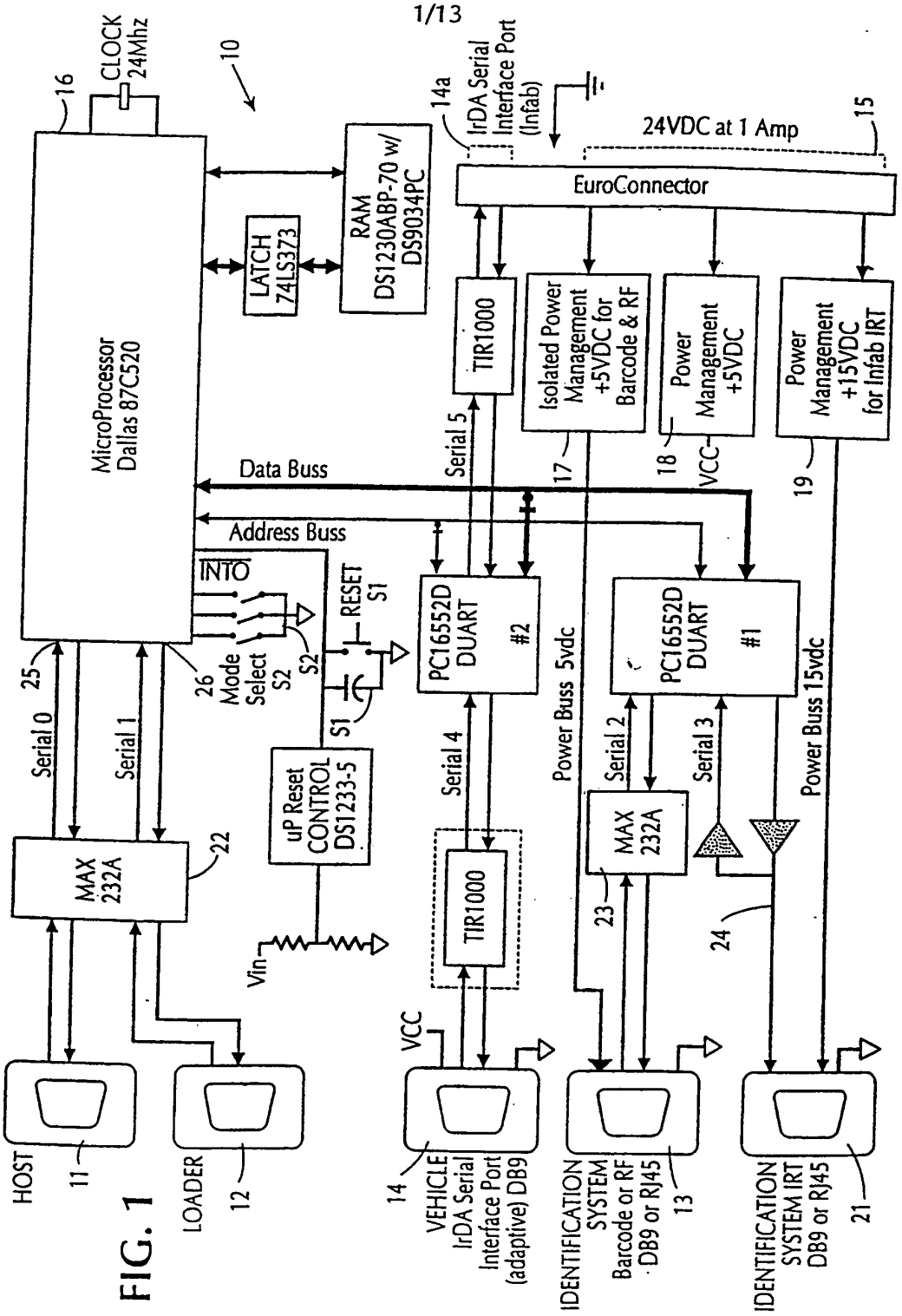

CLOCK 24Mhz
16
10
14a
IrDA Serial Interface Port (Infab)
24VDC at 1 Amp
15
EuroConnector
RAM DS1230ABP-70 w/ DS9034PC
LATCH 74LS373
MicroProcessor Dallas 87C520
TIR1000
Isolated Power Management +5VDC for Barcode & RF
Power Management +5VDC
Power Management +15VDC for Infab IRT
17
18
VCC
19
Serial 5
Data Buss
Address Buss
INTO
RESET S1
S1
S2
Mode Select S2
26
25
Serial 0
Serial 1
PC16552D DUART #2
PC16552D DUART #1
Serial 4
Serial 2
Serial 3
Power Buss 5vdc
Power Buss 15vdc
uP Reset CONTROL DS1233-5
MAX 232A
22
MAX 232A
23
24
Vin
TIR1000
VCC
HOST
11
LOADER
12
14
VEHICLE IrDA Serial Interface Port (adaptive) DB9
IDENTIFICATION SYSTEM Barcode or RF DB9 or RJ45
13
IDENTIFICATION SYSTEM IRT DB9 or RJ45
21

FIG. 1

FIG. 2

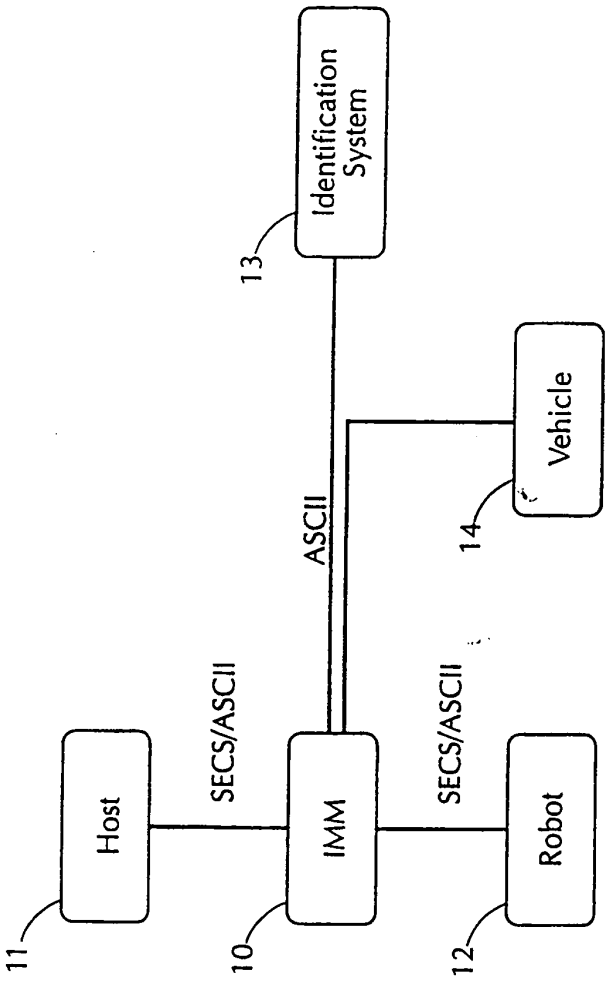
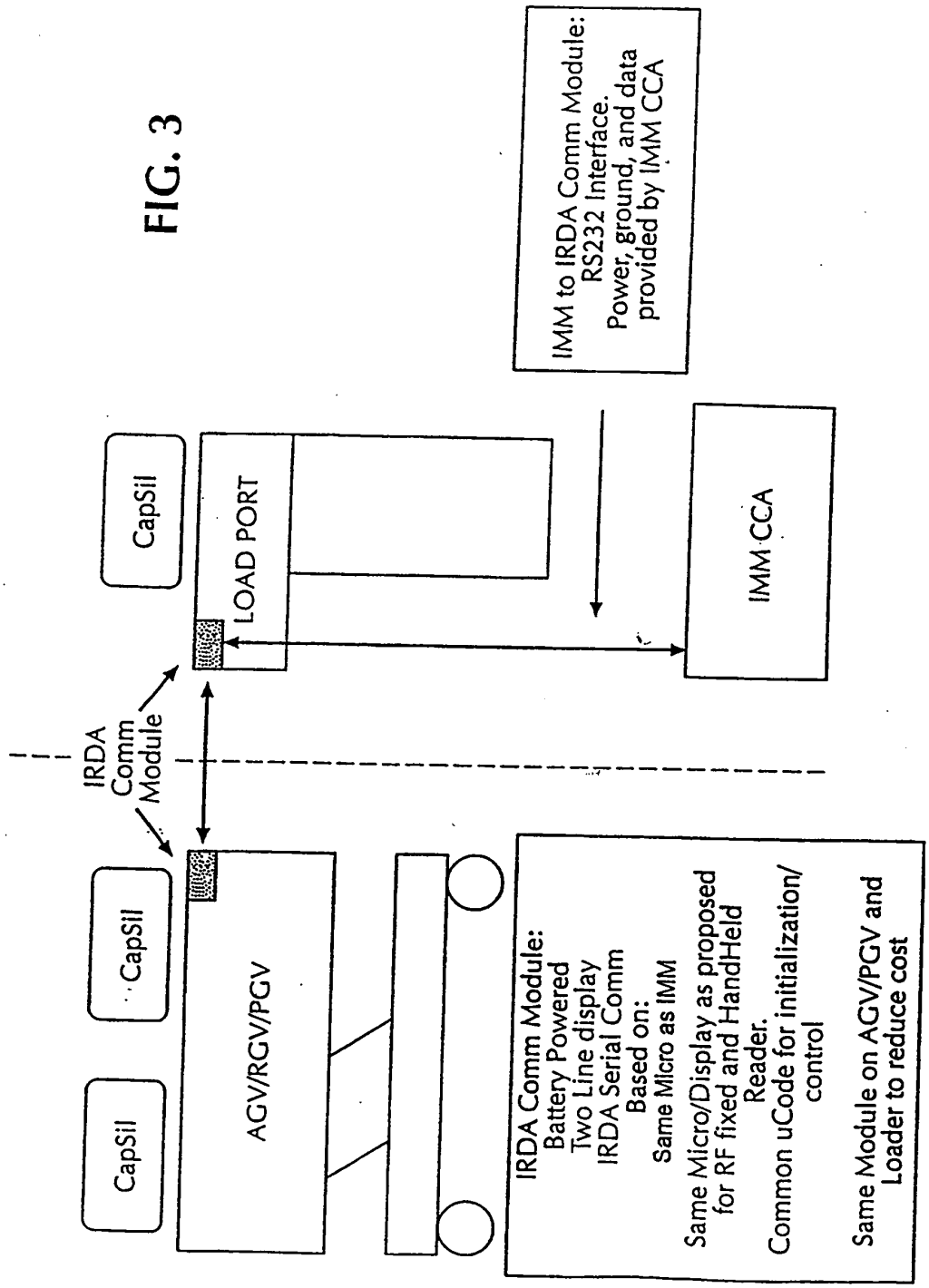

Identification System
13
ASCII
Vehicle
14
SECS/ASCII
SECS/ASCII
Host
IMM
Robot
11
10
12

FIG. 3


IMM to IRDA Comm Module:
RS232 Interface.
Power, ground, and data
provided by IMM CCA
IMM CCA
LOAD PORT
CapSil
IRDA
Comm
Module
CapSil
CapSil
AGV/RGV/PGV
IRDA Comm Module:
Battery Powered
Two Line display
IRDA Serial Comm
Based on:
Same Micro as IMM
Same Micro/Display as proposed
for RF fixed and HandHeld
Reader.
Common uCode for initialization/
control
Same Module on AGV/PGV and
Loader to reduce cost

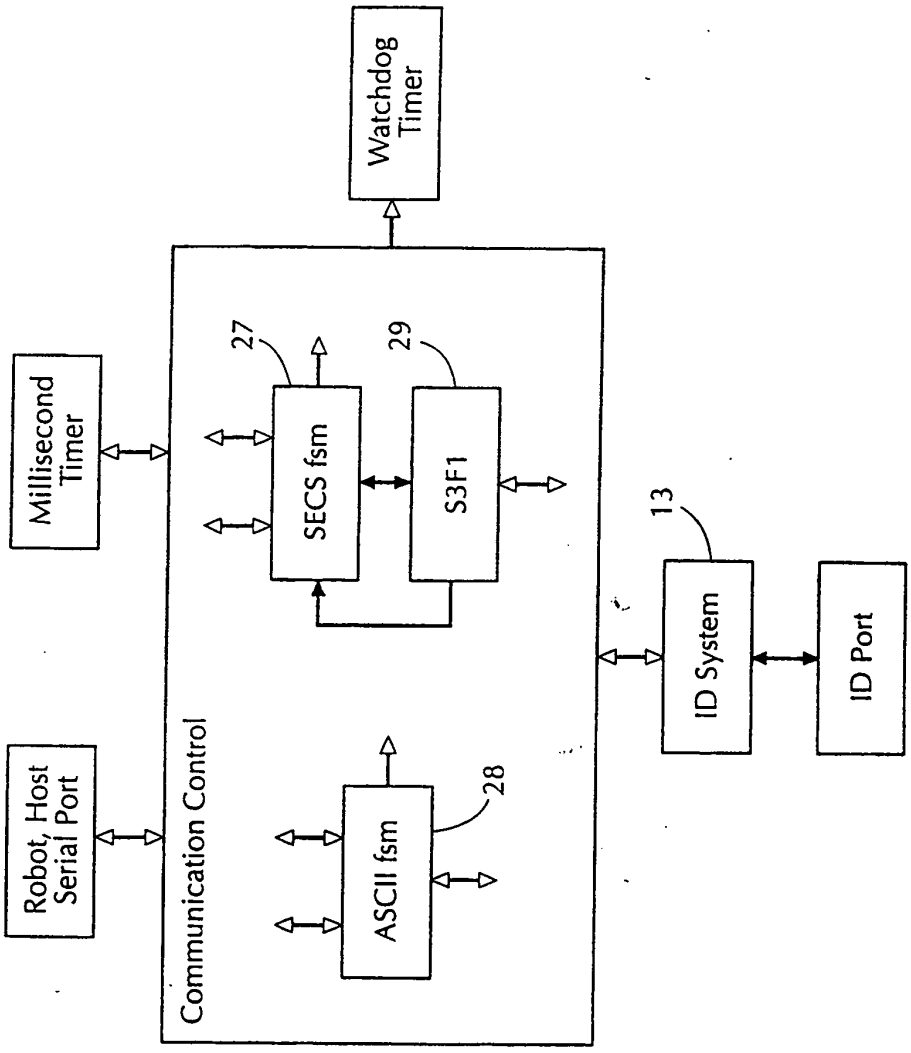

Watchdog Timer
Millisecond Timer
27
SECS fsm
29
S3F1
13
ID System
ID Port
Robot, Host Serial Port
Communication Control
ASCII fsm
28

FIG. 4

FIG. 5

Message from Robot to Host

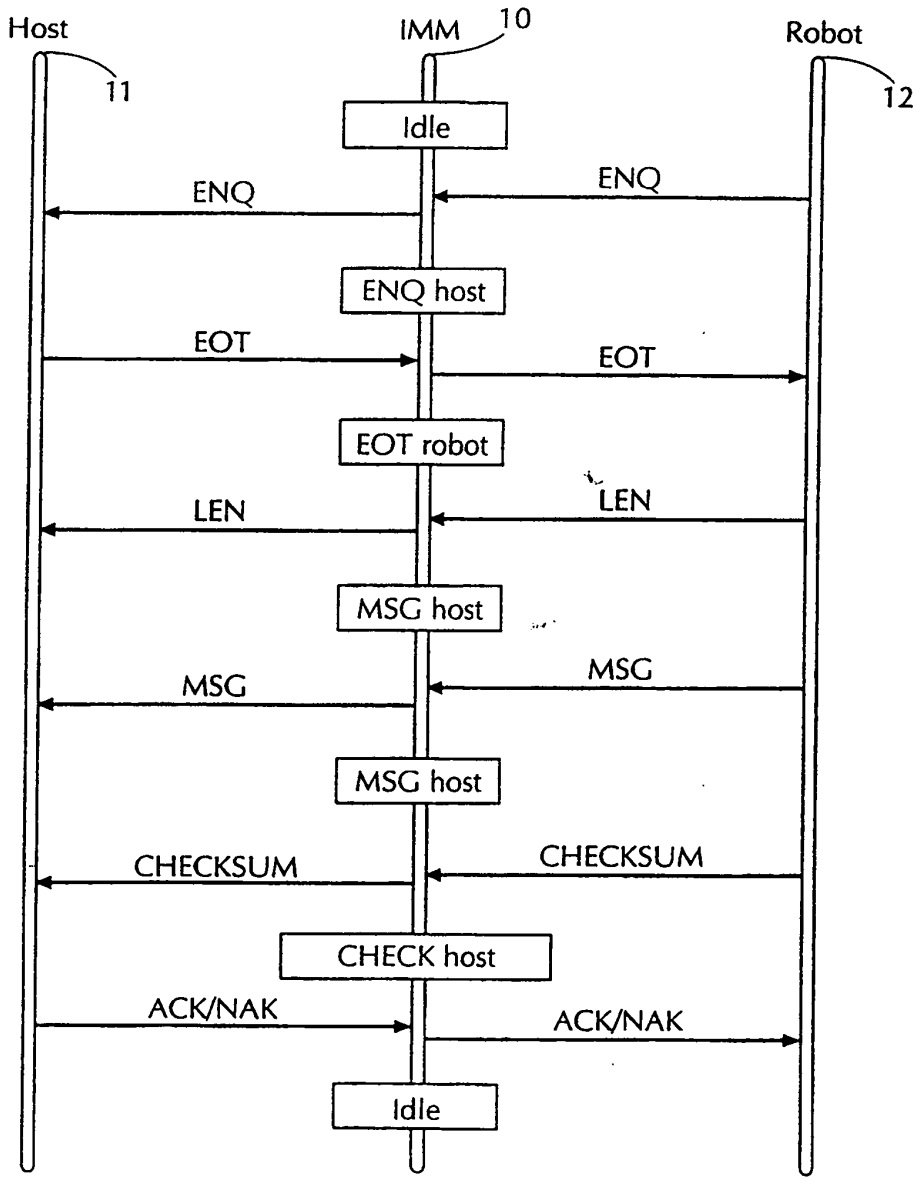


FIG. 6

Message from Robot

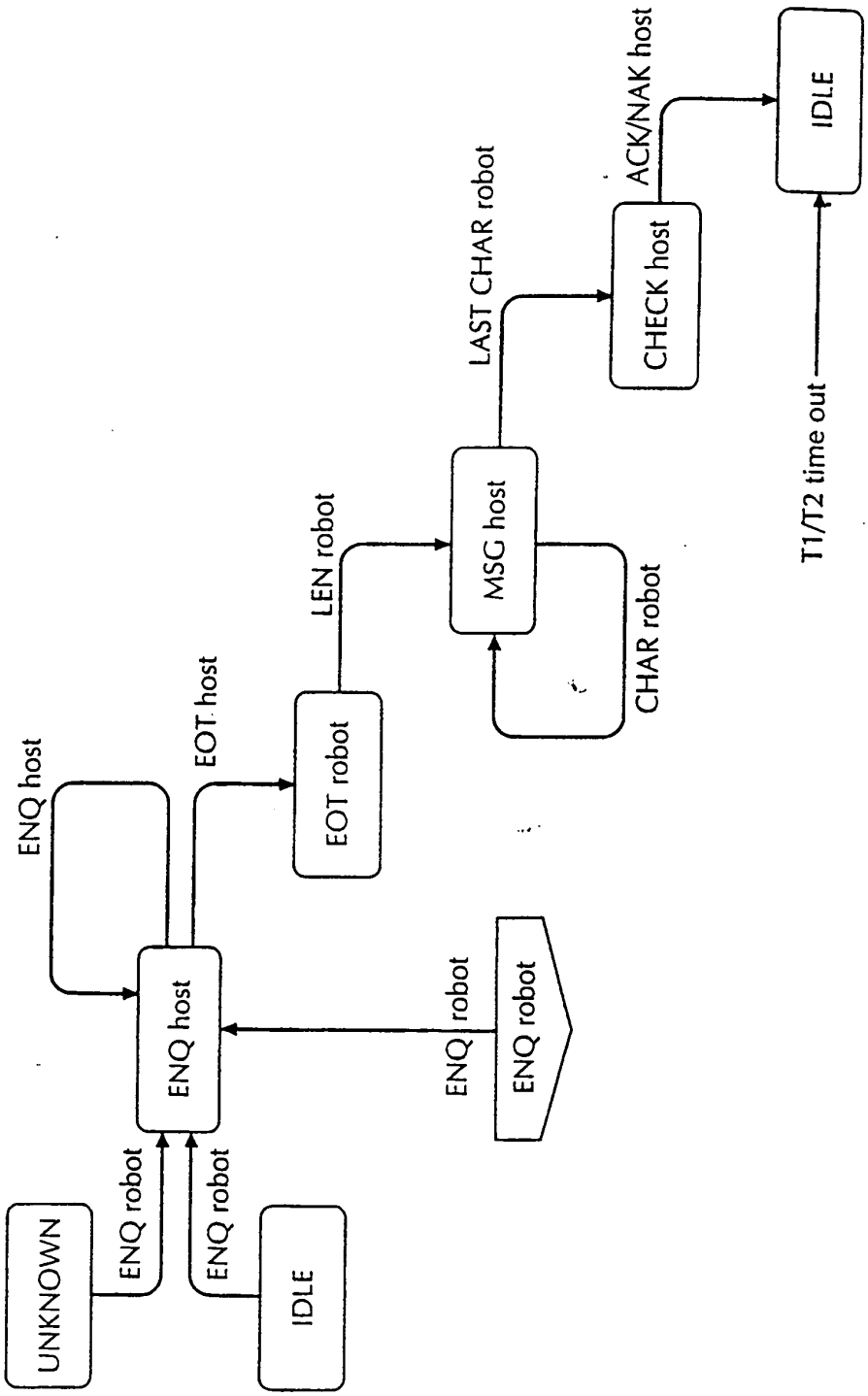


FIG. 7

UNKNOWN State Resolution

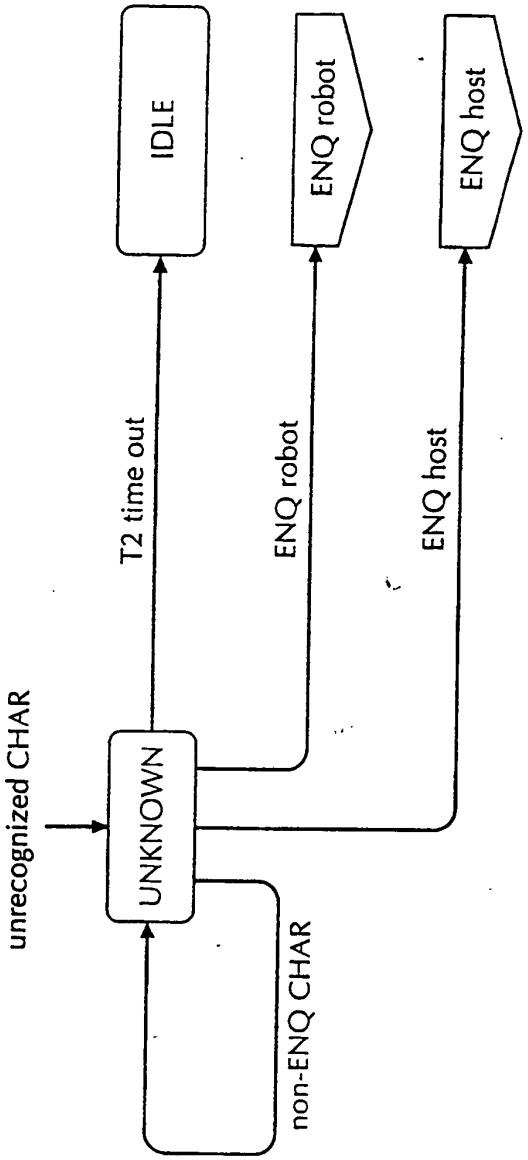


FIG. 8

Message from Host to Robot

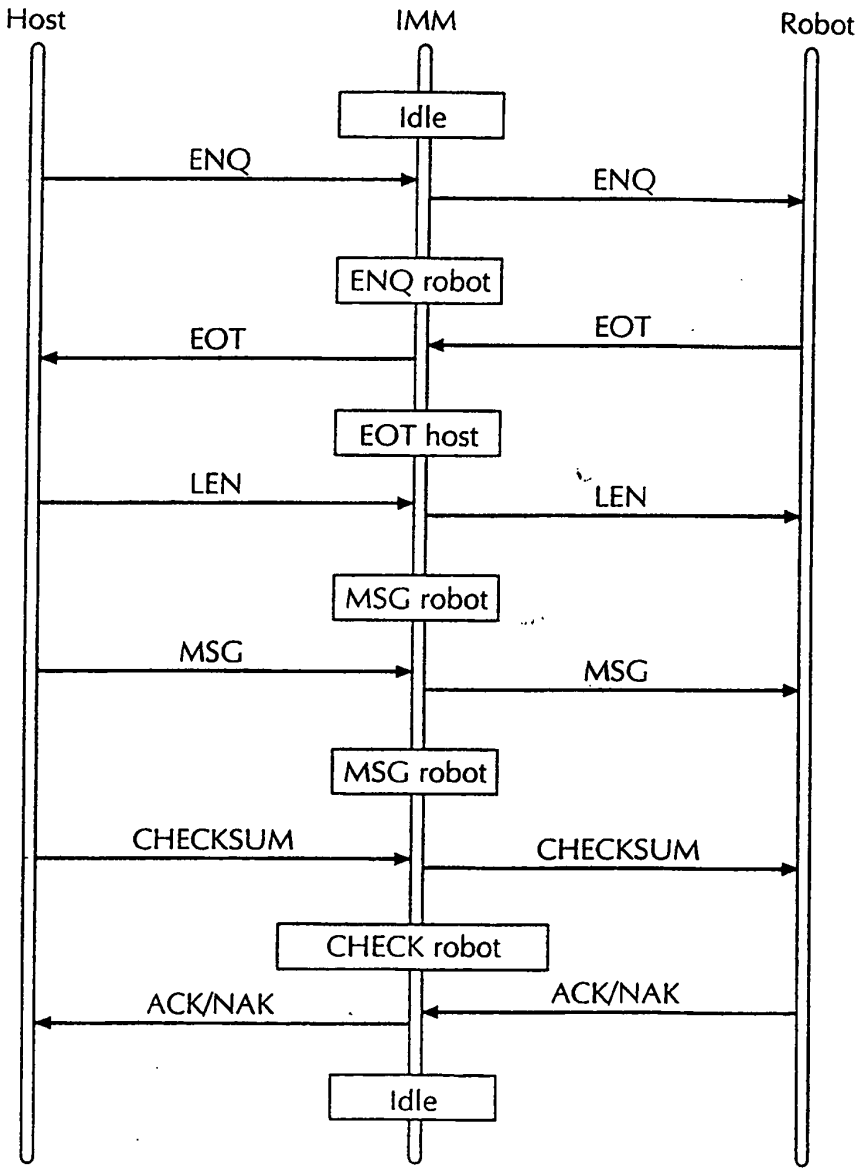
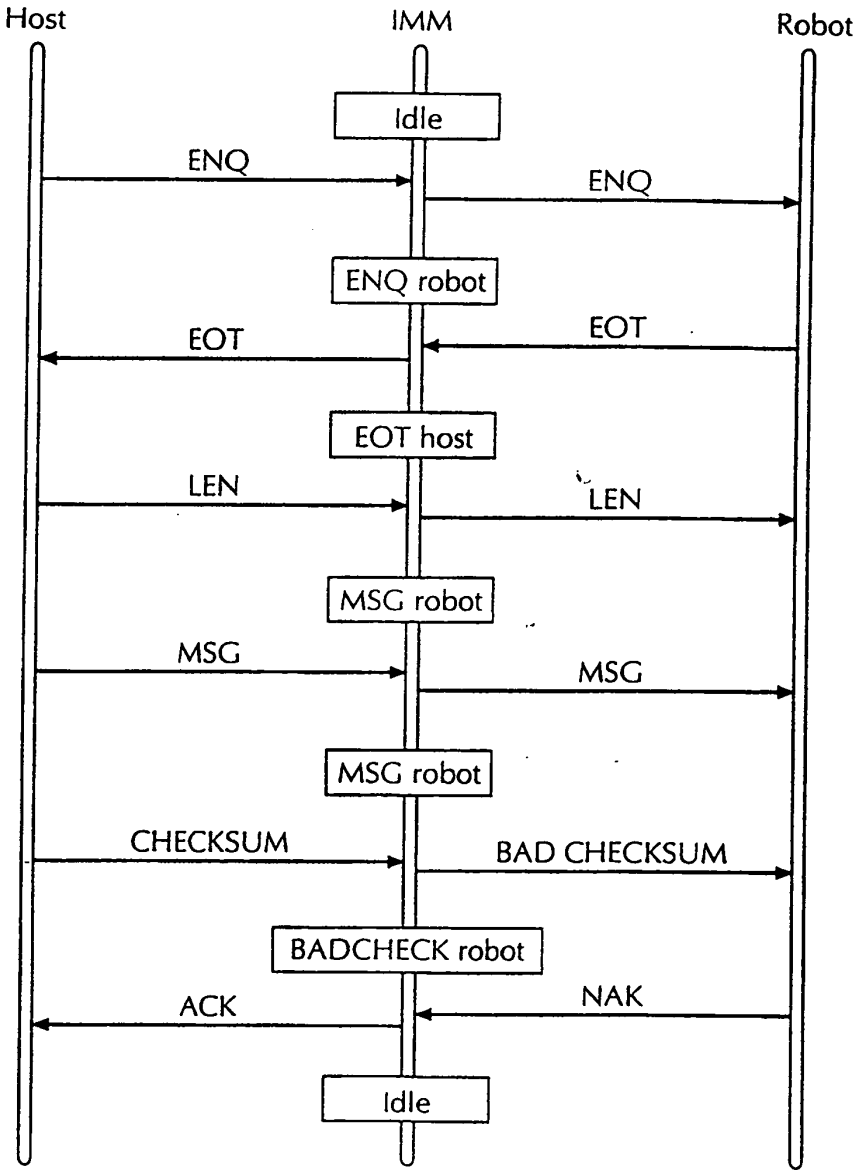


FIG. 9

Message from Host to IMM



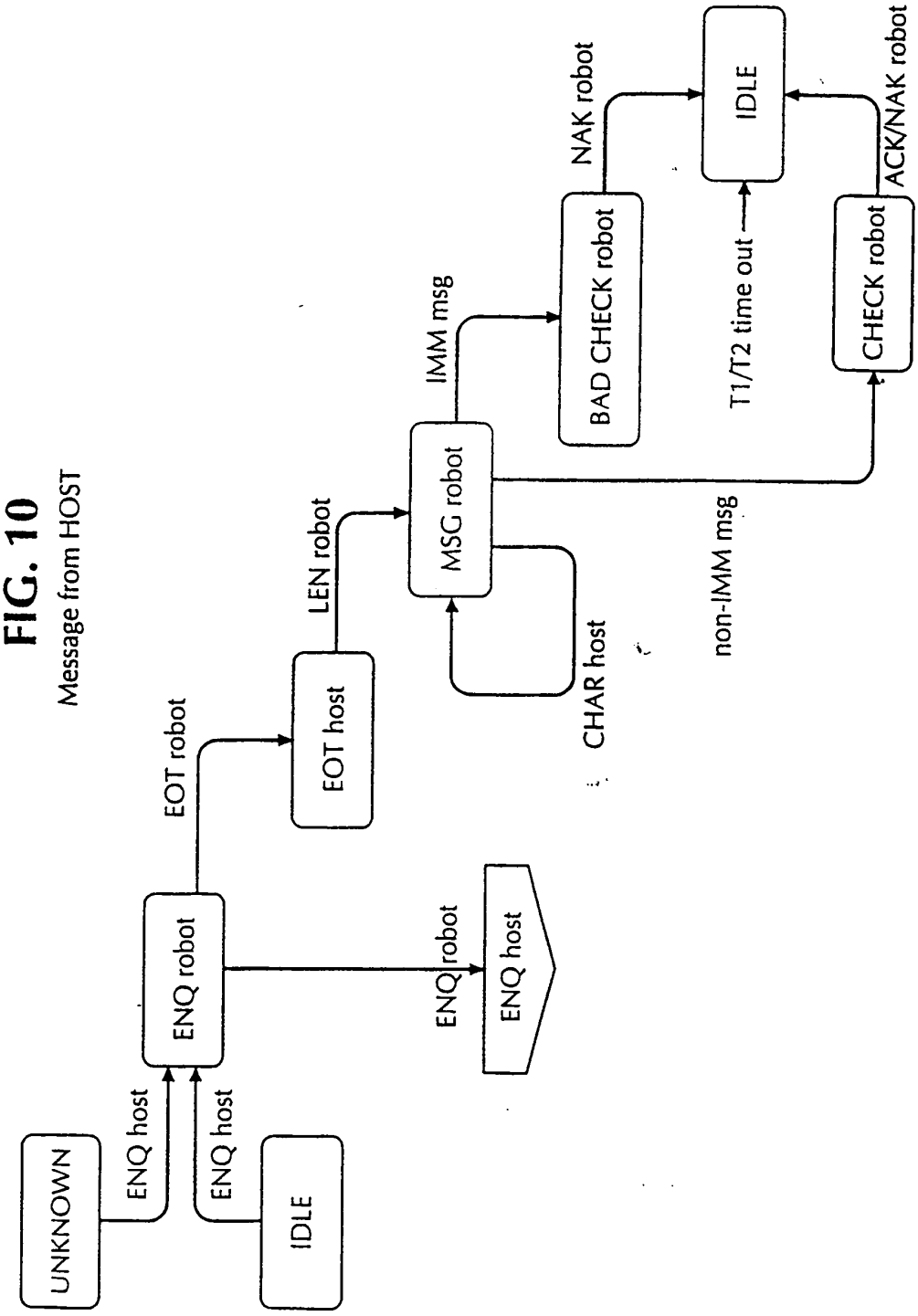


FIG. 10
Message from HOST

FIG. 11

Message from IMM to Host

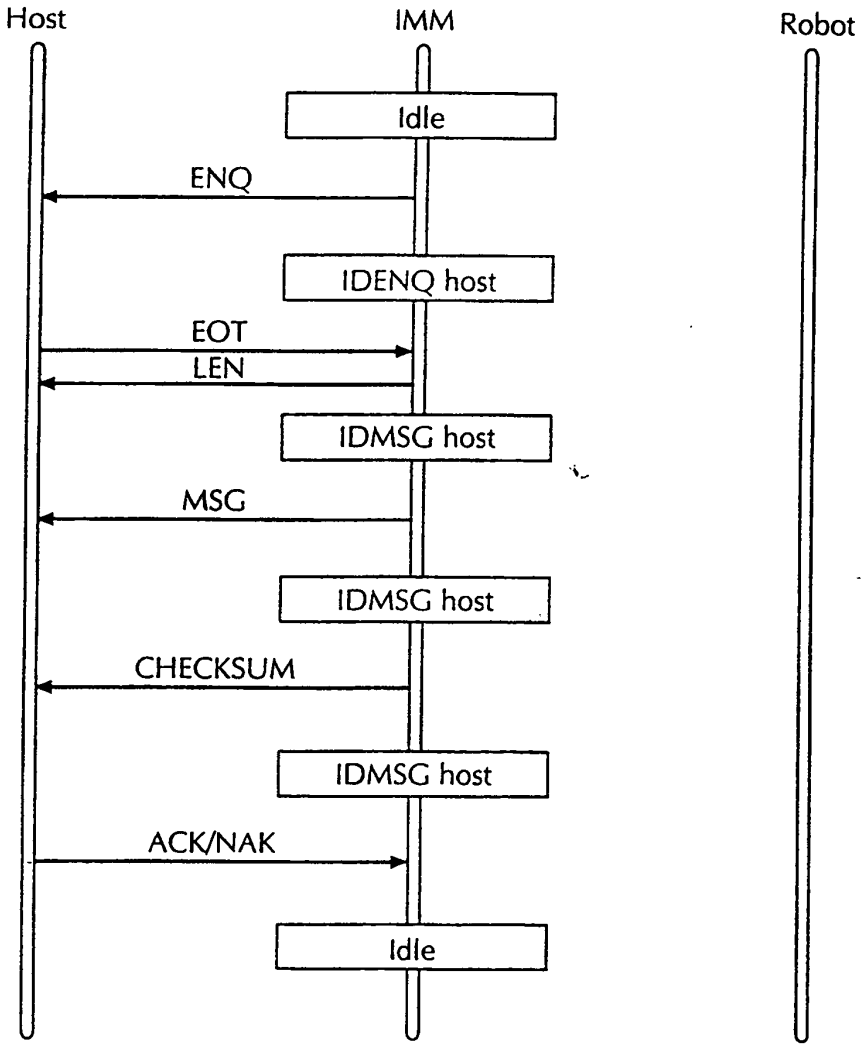


FIG. 12

Message from IMM

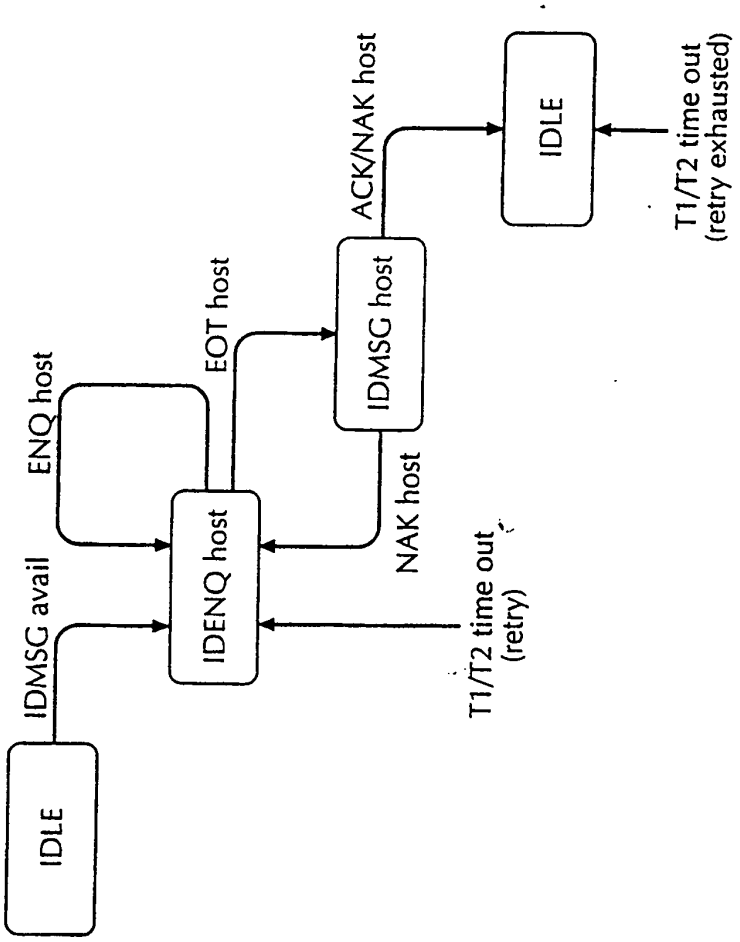
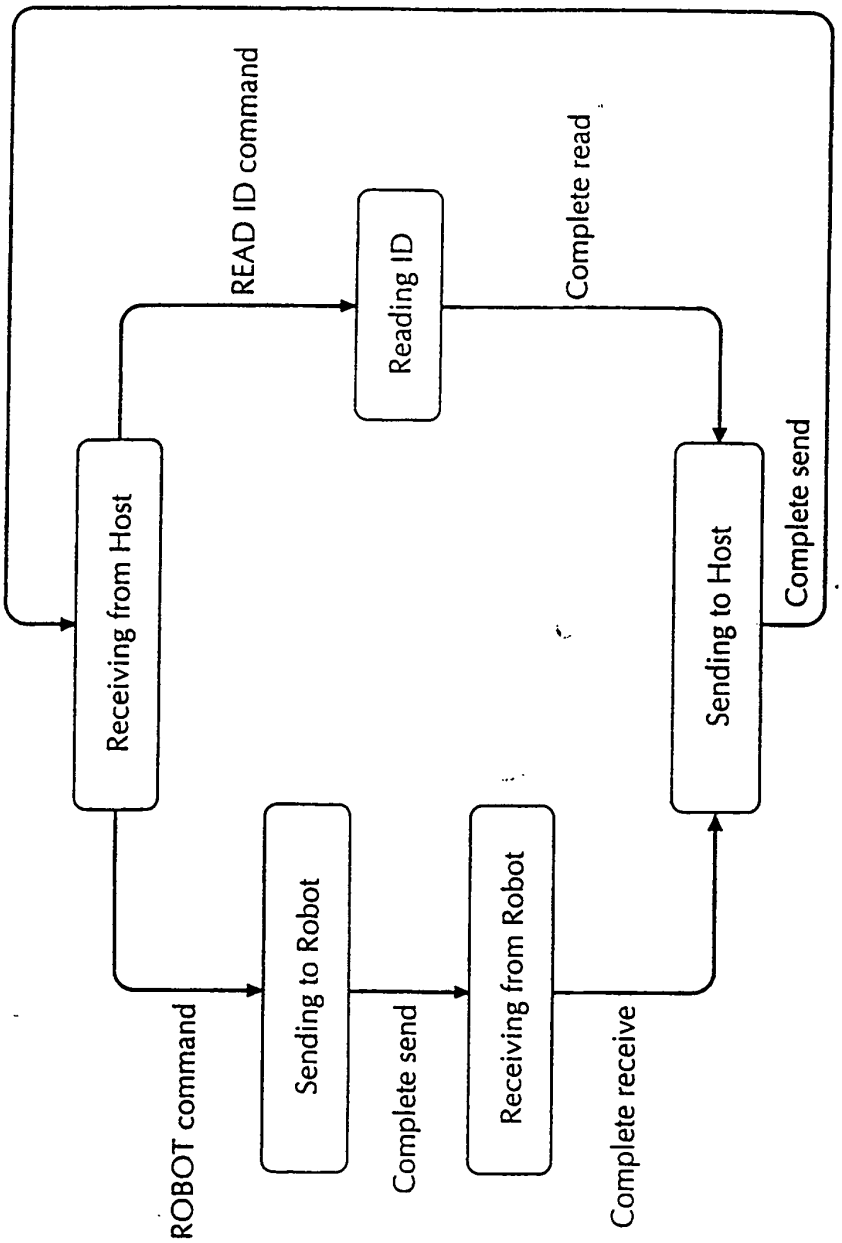


FIG. 13


Receiving from Host
READ ID command
Reading ID
Complete read
Sending to Host
Complete send
ROBOT command
Sending to Robot
Complete send
Receiving from Robot
Complete receive